Models of Computation: Automata and Processes

J.C.M. Baeten

January 4, 2010

Preface

Computer science is the study of discrete behaviour of interacting information processing agents.

Here, *behaviour* is the central notion. Computer science is about *processes*, things happening over time. We are concerned with an agent that is executing actions, taking input, emitting output, communicating with other agents. Moreover, this behaviour is *discrete*, takes place at separate moments in time, and we can observe separate things happening consecutively. Continuous behaviour is treated by dynamics or mechanics in physics, and does not belong to computer science. (However, when we talk about software control of a machine, the interplay of continuous behaviour and discrete behaviour becomes important.) Thus, the study of analog computers is no part of computer science.

An *agent* is simply an entity that can act. Primarily, we think of a computer executing a piece of software, but an agent can also be a human being when we consider his (her) behaviour discretely. Techniques from computer science are increasingly applied in systems biology, where we consider behaviour of a living cell or components of a cell. Also, an organisation or parts of an organisation can be considered as agents, for instance in workflow. In mechanical engineering, the behaviour of a machine can be modeled discretely. The study of components of an agent, their structure and their design can also be considered part of computer science.

Interaction is also a central notion in computer science. Agents interact amongst themselves, and interact with the outside world. Usually, a computer is not stand-alone, with limited interaction with the user, executing a batch process, but is always connected to other devices and the world wide web.

Information is the stuff of computer science, the things that are processed, transformed, sent around.

Given this definition of computer science, we can explain familiar notions in terms of it. A computer program is a prescription of behaviour, by transformation it can generate specific behaviour. An algorithm is a description of behaviour. In this book, we will give a much more precise definition of what an algorithm is. Computation refers to sequential behaviour, not considering interaction. Also this notion will be explained much more precisely here. Communication is interaction with information exchange. Data is a manifestation of information. Intelligence has to do with a comparison between different agents, in particular between a human being and a computer.

This book is an introduction to the foundations of computer science, suitable for teaching in any undergraduate program in computer science or related fields. It studies discrete behaviour, both with and without taking interaction into account. It presents an abstract model of discrete behaviour, called an *au-*

tomaton or *transition system*. An abstract model is a mathematical structure that captures the essential features, leaving out all irrelevant detail.

A more elaborate abstract model is that of the Turing machine. This model provides us with the notion of computability and the notion of an algorithm. These are central notions in computer science: a problem class that is computable can in principle be calculated by a computer, provided certain limitations on speed and memory are met; on the other hand, a problem class that is non-computable can never be solved on any computer, no matter how fast it is or how much memory it has. Thus, we can assert what a computer can do. More importantly, we can assert what a computer cannot do.

The classical Turing machine model has an important drawback: it does not consider interaction. Thus, a computer is studied as a stand-alone machine, with no connections to other computers and the world, and a computer program can be seen as a function that transforms input into output. Adding interaction leads to a modification of a Turing machine, the so-called Interactive Turing machine. Alongside the notion of a computable function, this gives us the notion of an executable process.

Important notions: formal language and communicating process, automaton and transition system, language equivalence and bisimulation, grammar and recursive specification, Turing machine, Chomsky hierarchy.

Main references for the part about formal languages and automata theory are [10], [9] [14] and [1], main reference for the part about process theory is [2] (see also [11], but also older textbooks [8], [12] can be consulted). More specialized references are [3], [13], [4], [6], [7], [5].

Contents

Chapter 1

Introduction

What makes something a computer? When do we call something a computer? Well, color, size and brand do not matter. Nor is it important whether it has a monitor, a mouse or a keyboard.

Figure 1.1: Abstract model of a computer.

Figure 1.1 presents an abstract model of a computer. It contains the following ingredients:

1. A *finite control*. A computer is in essence *discrete*: it evolves not by continuous change along a trajectory (as a physical phenomenon), but by a discrete change from one state to the next. Thus, the control has a finite number of *states*, modes it can be in. Each time the computer does something, performs an elementary action or instruction, it can change state, go from a given state to another state (or the same state). The control may be finite, but not a priori *bounded*: we can assume we have as many states as necessary in order to tackle a given problem.

2. The *memory* is a facility where things can be put and retrieved. There are always a finite number of things in memory, but the memory is not *bounded*: there is always room to put more things, if this is needed.

3. The control and the memory *interact*, can exchange information. The control can store something in the memory, and can retrieve something from the memory.

4. The control *interacts* with the external world. This external world is not part of the computer, and includes a user, other computers, the world wide web. From the external world, something can be input into the computer, and from the computer, something can be output to the external world.

Several times in this enumeration, the notion of a *thing*, an *action*, *instruction* or a *datum*, a piece of *information* is used, the 'stuff' that is sent around between different parties. We do not specify what this 'stuff' is exactly, but just assume throughout some given non-empty finite set $\mathcal{A}$ called the *alphabet*, which is not further specified. In a similar vein, we assume a given non-empty finite set of states called $\mathcal{S}$. The set of states contains an *initial* state, the state at startup, and a subset of *final* states, where termination can take place.

A finite control over a given alphabet and a given set of states with an initial state and a set of final states is called a *finite (non-deterministic) automaton*. The memory will be modeled by a particular process, the *Turing tape*. The tape holds a sequence of things called *data*, and at every moment is active at an element of the sequence, the *focus*. At the focus, the datum can be output or a new datum can be input, or the focus can shift one element to the right or left.

Now the abstract model to be considered in this book is the model of the *Interactive Turing machine*: there is a finite control denoted as an automaton, interacting with the external world, and a memory denoted as a Turing tape, interacting with the control. Schematically, this is presented in Figure 1.2.

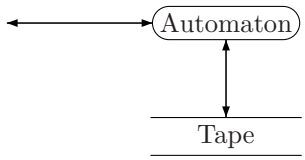

Figure 1.2: Interactive Turing machine.

This is the model that concerns the part of this book about processes. In the part about formal languages, we simplify this picture even further. We do this by hiding the notion of interaction.

First, we limit the interaction with the external world by only allowing input at the initial state, and only allowing output at a final state. This means we consider the computer as a stand-alone device, that does not interact with the external world during computation. We consider the computer as a device that transforms input into output, a *function* from input to output. Thus, we have a *batch process* such as they existed before the advent of the terminal. Finally, we still have the interaction with the memory, but will hide this interaction in our description.

Due to these limitations, it is possible to assume that input and output consist of a *finite sequence* of actions or data, a *string*. The content of the memory will also be a string of actions at any given time (but it can grow or shrink over time). This leads to the abstract model of Figure 1.3, which is called the *classical Turing machine*.

When interaction is involved, this picture of separating input and output is not adequate any longer. Consider an airline reservation system. It is not the case that a user interacting with this system will have a series of actions ready at the start of the interaction, rather, the user will have an initial action, the system responds with a webpage containing a number of choices, the user will select a number of options, the system reacts, etc. This means a series of

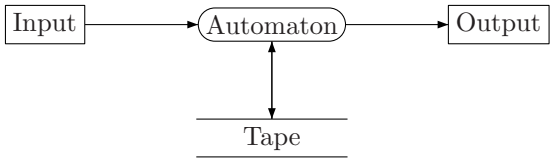

Figure 1.3: Classical Turing machine.

actions as input eventually leading to a series of actions as output is not adequate to describe this interaction. In particular, the airline reservation system may not have a final answer, it may be always on. Thus, it is important to keep information about actions, even when these actions do not lead to a final state. Moreover, it is also important to keep track of the moments of choice passed in the course of the interaction. Therefore, considering interaction, we will use the interactive Turing machine rather than the classical Turing machine.

In the next chapter, we simplify Figures 1.2 and 1.3 even further, by leaving out the memory altogether.

We use the following notations: elements of $\mathcal{A}$ are denoted with lower case letters from the beginning of the alphabet $a, b, c, \ldots$. A finite sequence of elements of $\mathcal{A}$ is called a *string* or a *word* and denoted by juxtaposition, so e.g. $aabb$ and bca are strings. If w, v are two strings, then wv is the concatenation of the two strings, the string consisting of w followed by v, so if $w = aabb$ and $v = bca$ then $wv = aabbbca$. The empty string consisting of no letters is denoted ε, and the set of all strings is denoted $\mathcal{A}^*$. This set can be inductively defined as follows:

1. (basis) $\varepsilon \in \mathcal{A}^*$, the empty string is a string;
2. (step) For all $a \in \mathcal{A}$ and all $w \in \mathcal{A}^*$, the string that starts with a and follows with the string w is a string, $aw \in \mathcal{A}^*$.

Every string can be made in this way, starting from the empty string, and each time adding one element of $\mathcal{A}$ in the front. Adding an element of $\mathcal{A}$ in front of a string is called *prefixing* the string with this element. Instead of $a\varepsilon$, we often write just a for the string consisting of just the letter a.

Concatenation is inductively defined as follows:

1. If $w = \varepsilon$, then $wv = v$;
2. If $w = aw'$, then $wv = a(w'v)$.

A subset L of $\mathcal{A}^*$ is called a *language*.

The *length* of a string is the number of elements it contains, and is defined inductively as follows:

1. $|\varepsilon| = 0$;
2. For all $w \in \mathcal{A}^*$ and $a \in \mathcal{A}$ we have $|aw| = |w| + 1$.

A string v is a *prefix* of string w iff there is a string u such that $vu = w$, and v is a *substring* of w iff there are strings u, x such that $xvu = w$. The number of a's in w, $\#_a(w)$, can again be defined inductively:

1. $\#_a(\varepsilon) = 0$;
2. For all $w \in \mathcal{A}^*$ we have $\#_a(bw) = \#_a(w)$ if $a, b \in \mathcal{A}, a \neq b$, and $\#_a(aw) = \#_a(w) + 1$.

The *reverse* of a string w, denoted w^R is the string read backwards. Defined inductively:

1. $\varepsilon^R = \varepsilon$;
2. For all $w \in \mathcal{A}^*$ and $a \in \mathcal{A}$ we have $(aw)^R = (w^R)a$ (the concatenation of string w^R and string a).

Exercises

1.0.1 Prove by induction on $|u|$ that for all strings $u, v \in \mathcal{A}^*$ we have $|uv| = |u| + |v|$.

1.0.2 Define for string $u \in \mathcal{A}^*$ and natural number n the power u^n by induction on n:

(a) $u^0 = \varepsilon$;

(b) $u^{n+1} = uu^n$.

Use induction on n to show that $|u^n| = n|u|$ for all $u \in \mathcal{A}^*$ and $n \geq 0$.

1.0.3 Prove that $(uv)^R = v^R u^R$ for all strings $u, v \in \mathcal{A}^*$. Also, prove $(w^R)^R = w$ for all strings $w \in \mathcal{A}^*$.

1.0.4 The set of strings $\{w\}^*$ consists of the string w repeated an arbitrary number of times n ($n \geq 0$). Prove $\{a\}^* = \{a^n \mid n \geq 0\}$. Find four elements of $\{ab\}^*$.

Chapter 2

Finite Automata

2.1 Automata and languages

Starting from the abstract model of the Turing machine in the previous chapter, we remove the memory and the interaction with the memory, and we just have a finite control that transforms the input (an element of $\mathcal{A}^*$) into output (again, an element of $\mathcal{A}^*$).

We make one further simplification: we limit the set of outputs to yes or no. If a certain input string leads to yes, we say the machine *accepts* the string. See Figure 2.1.

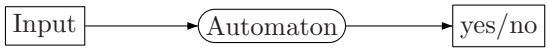

Figure 2.1: Abstract model of an automaton.

An automaton has a given set of states $\mathcal{S}$, a given alphabet $\mathcal{A}$, a transition relation that explains how to get from one state to the next state, an initial state and a set of final states.

Definition 2.1 (Automaton)**.** An *automaton* M is a quintuple $(\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \downarrow)$ where:

1. $\mathcal{S}$ is a finite set of states,
2. $\mathcal{A}$ is a finite alphabet,
3. $\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the set of *transitions* or *steps*,
4. $\uparrow \in \mathcal{S}$ is the initial state,
5. $\downarrow \subseteq \mathcal{S}$ is the set of final states.

If $(s, a, t) \in \rightarrow$, we write $s \xrightarrow{a} t$, and this means that the machine, when it is in state s, can consume input symbol a and thereby move to state t. If s is a final state, $s \in \downarrow$, we write $s \downarrow$.

Writing $s \xrightarrow{a,b} t$ means that both $s \xrightarrow{a} t$ and $s \xrightarrow{b} t$.

Notice that it is possible that there are states $s, t, u \in \mathcal{S}$ and $a \in \mathcal{A}$ with $s \xrightarrow{a} t$ and $s \xrightarrow{a} u$ and $t \neq u$. Occurrence of this is called *non-determinism*. It means that from a given state, consuming a may lead to different states, the outcome is not determined. It reflects uncertainty about the exact circumstances of a step, or hiding of the details of determination (just like the input of a string into a search engine will never give the same answer).

Notice that between two given states, there is *at most one* step with label a, such a step either exists or not.

Example 2.2. Let us consider an example. The automaton in Figure 2.2 has the initial state labeled by a small incoming arrow, and the final states labeled by small outgoing arrows. All transitions are labeled by $a \in \mathcal{A}$. Notice that names of states are often not given as they cannot be observed. We will consider two automata that only differ in the naming of states or the layout to be the same. Stated formally, we say we consider isomorphic automata to be the same. We will not define isomorphism on automata, instead, we will address more extensively when two automata are considered the same in the next section.

Notice that from the initial state, there are two different outgoing transitions with the same label, which means there is non-determinism: executing a from the initial state may lead to two different states. Notice that the state on the bottom right can never be reached from the initial state by executing transitions. Such a state, and its attached transitions, will be called unreachable. Unreachable states and transitions will not play a role in the following considerations.

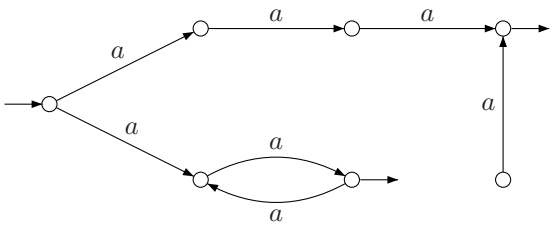

Figure 2.2: Example automaton.

We make precise the notion of a reachable state in an automaton.

Definition 2.3. Let automaton $M = (\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \downarrow)$ be given. We generalize the transition relation $\rightarrow$ to paths, by an inductive definition. Let $w \in \mathcal{A}^*$ be a string, and $s, t \in \mathcal{S}$. Then $s \xrightarrow{w}\!\!\!\!\rightarrow t$ denotes a *path* from s to t with *label* w, and $s \xrightarrow{w}\!\!\!\!\rightarrow t$ holds exactly when it can be derived by means of the following clauses:

1. For all $s \in \mathcal{S}$, we have $s \xrightarrow{\varepsilon}\!\!\!\!\rightarrow s$;
2. For all $s, t, u \in \mathcal{S}$, if $s \xrightarrow{a} t$ and $t \xrightarrow{w}\!\!\!\!\rightarrow u$, then $s \xrightarrow{aw}\!\!\!\!\rightarrow u$.

If there is a path from s to t, we say t is *reachable* from s, and write $s \longrightarrow\!\!\!\!\rightarrow t$. A state s is *reachable* in M iff $\uparrow \longrightarrow\!\!\!\!\rightarrow s$.

For example, in the automaton shown in Figure 2.2, if s is the final state on the top right, then $\uparrow \stackrel{aaa}{\longrightarrow\!\!\!\!\!\rightarrow} s$, and if t is the final state on the bottom, then $\uparrow \stackrel{aa}{\longrightarrow\!\!\!\!\!\rightarrow} t$, $\uparrow \stackrel{aaaa}{\longrightarrow\!\!\!\!\!\rightarrow} t$, etc. In general, $\uparrow \stackrel{a^n}{\longrightarrow\!\!\!\!\!\rightarrow} t$ for every even $n > 0$.

If we have given a certain automaton, then it will accept those input strings for which there is a path from the initial state to a final state with this series of letters as labels, consecutively. So the automaton in Figure 2.2 accepts the strings aaa and a^n for every even $n > 0$, and no others, so the set of strings $\{a^n \mid n > 0, n \text{ is even or } n = 3\}$. We make this precise in the following definition.

Definition 2.4. The automaton M *accepts* the string w, if there is a state $s \in \mathcal{S}$ with $\uparrow \stackrel{w}{\longrightarrow\!\!\!\!\!\rightarrow} s$ and $s \downarrow$. In this case we also say the path from $\uparrow$ to s is a *trace* of M.

Finally, the *language* of M is the set of all strings accepted by M, $\mathcal{L}(M) = \{w \in \mathcal{A}^* \mid \exists s \in \mathcal{S} : \uparrow \stackrel{w}{\longrightarrow\!\!\!\!\!\rightarrow} s, s \downarrow\}$.

A consequence of this definition is that if a string w is not in the language of an automaton M, then it holds that whenever w is the label of a path of the automaton, then either it does not start from the initial state or it does not end in a final state. But it can also happen that w is not the label of any path of the automaton.

Example 2.5. We give another example. Consider the automaton M in Figure 2.3. Then its language is $\mathcal{L}(M) = \{a^n b \mid n \geq 0\}$. In the initial state, the action a can be executed an arbitrary number of times, remaining in the same state, until at some time the edge labeled b is taken to the final state. The right-most edge, labeled a, has no influence on the language at all, since taking it can never lead to a final state. The right-most state is a so-called *deadlock* state, where no activity whatsoever is possible.

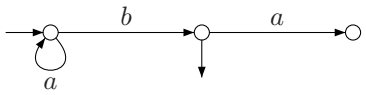

Figure 2.3: $L = \{a^n b \mid n \geq 0\}$.

Thus, given an automaton, we can determine its language. It is also interesting to consider the reverse direction: given a language, find an automaton that accepts this language. We start with a simple example.

Example 2.6. Consider the set of all strings over $\mathcal{A} = \{a, b\}$ that start with the prefix ab. To find an automaton that accepts this language, just make a path from the initial state to a final state with label ab. Upon reaching this final state, allow any step, i.e., add a step with label a and label b from this state to itself. See Figure 2.4.

Example 2.7. To give a more complicated example, consider the set of all strings over $\mathcal{A} = \{a, b\}$, that do not contain a substring aab, so $L = \{w \in \mathcal{A}^* \mid aab \text{ is not a substring of } w\}$. To begin with, trace out the string aab from the initial state, but now make all the states *except* the right-most state final states,

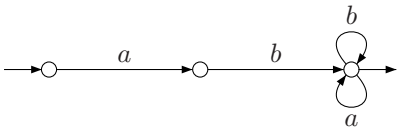

Figure 2.4: $L = \{w \in \mathcal{A}^* \mid ab \text{ a prefix of } w\}$.

see Figure 2.5. The left-most state denotes the state where no initial part of a string possibly leading to aab has occurred. There, any number of b's can take place, but as soon as one a occurs, we move one state to the right. There, executing a b cannot lead to the string aab any longer, so we move left again, but executing an a leads to a state where two consecutive a's have occurred. There, executing a b leads to deadlock (the b-step could also have been left out), but executing an additional a makes no difference: still, two consecutive a's have occurred, and the next step cannot be b.

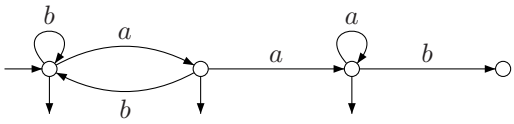

Figure 2.5: $L = \{w \in \mathcal{A}^* \mid aab \text{ is not a substring of } w\}$.

Thus, we see that for a number of languages, we can find an automaton that accepts this language. However, this is by no means true for *all* languages, there is only a very limited number of things that can be done by a computer without a memory.

One thing that a computer without memory cannot do is counting. We illustrate this as follows. Take the alphabet $\mathcal{A} = \{a, b\}$ and consider $L = \{a^n b^n \mid n \geq 0\}$. We can draw a picture for this language as shown in Figure 2.6. Notice that the initial state is also a final state. This system can take in any number of a's, remember how many a's it has received, and then count off just as many b's. Thus, it can check whether the number of a's and the number of b's is the same, it can compare quantities. As it can remember any given number, we can call this machine a simple counter.

Notice that this is not an automaton, as the number of states is infinite. We call a system that is like an automaton, but may have infinitely many states a *transition system*. We claim there is no automaton that accepts this language.

Theorem 2.8. Let $L = \{a^n b^n \mid n \geq 0\}$. There is no automaton that accepts L.

Proof. Proof by contradiction, so suppose there is an automaton

$$M = (\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \downarrow)$$

that accepts L. Then for each $n \geq 0$, there is a path in M from $\uparrow$ to a final state with label $a^n b^n$. Fix such a path for each n, and call s_n the state this path

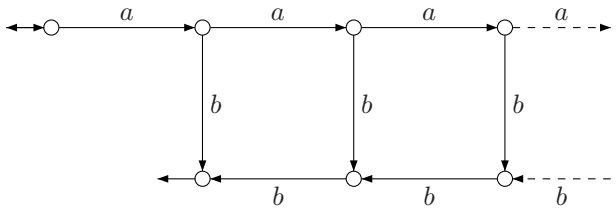

Figure 2.6: Simple counter.

is in after the a's and before the b's. This gives us states $s_0, s_1, \ldots, s_n, \ldots$ in $\mathcal{S}$. Since $\mathcal{S}$ is finite, at least two of these states must be the same, say $s_k = s_m$ for certain $k \neq m$. Then there is a path with label a^k from $\uparrow$ to s_k and a path with label b^m from s_m to a final state, from which it follows that there is a path with label $a^k b^m$ from $\uparrow$ to a final state, and $a^k b^m$ is accepted by M. This is a contradiction, so such an automaton M cannot exist. □

Later on, we will present a more general technique in order to prove that some language is not accepted by any automaton.

Definition 2.9. Let $L \subseteq \mathcal{A}^*$ be a language. L is called *regular* iff there is an automaton M that accepts L. If two automata M, M' accept the same language, $\mathcal{L}(M) = \mathcal{L}(M')$, we say the automata are *language equivalent*, and write $M \approx M'$.

The theorem above implies that $L = \{a^n b^n \mid n \geq 0\}$ is not regular. Earlier, we saw examples of languages that are regular. It will turn out that the set of regular languages is very limited, or, stated differently, computers without memory cannot do much.

Language equivalence is indeed an equivalence relation.

Theorem 2.10. Language equivalence is an equivalence relation on automata.

Proof.

1. Reflexivity: $M \approx M$ as $\mathcal{L}(M) = \mathcal{L}(M)$, so we have reflexivity for every automaton M.
2. Symmetry: if $M \approx M'$ for two automata M, M', then their languages are the same, so also $M' \approx M$, and we have symmetry.
3. Transitivity: if $M \approx M'$ and $M' \approx M''$, then $\mathcal{L}(M) = \mathcal{L}(M')$ and $\mathcal{L}(M') = \mathcal{L}(M'')$, so $\mathcal{L}(M) = \mathcal{L}(M'')$, which implies $M \approx M''$ so we have transitivity.

□

Example 2.11. Many real-life processes can be conveniently modeled by an automaton. A university that organises a colloquium could draw the following automaton for its internal process, see Figure 2.7. Registration starts by the receipt of a filled-in webform. It is checked whether or not this webform has all the necessary information. If it does not, the customer is asked to complete the

form. If it does, it is checked whether or not the registration has a VIP-code. If it does, the registration is complete and an invitation is sent. If it does not, the customer has to pay €150 before the early registration deadline, and €195 after this date. When the money is received, a code is sent to the customer and an invitation is sent.

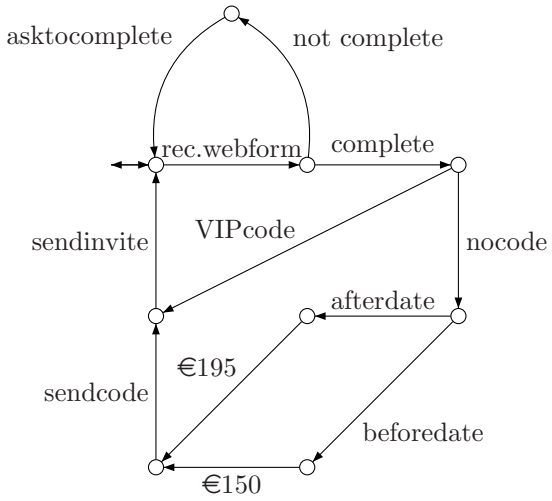

Figure 2.7: Registration process.

Example 2.12. The control flow of a program can be modeled as an automaton. Consider the following PASCAL program to calculate factorials:

```
PROGRAM factorial(input,output);
VAR i, n, f:  0..maxint;
BEGIN
     read(n);
     i := 0; f := 1;
     WHILE i < n DO
          BEGIN i := i + 1; f := f * i END;
     write(f)
END
```

The control flow of this program can be modeled as an automaton as shown in Figure 2.8.

Exercises

2.1.1 Let $\mathcal{A} = \{a, b\}$. Construct automata that accept exactly:

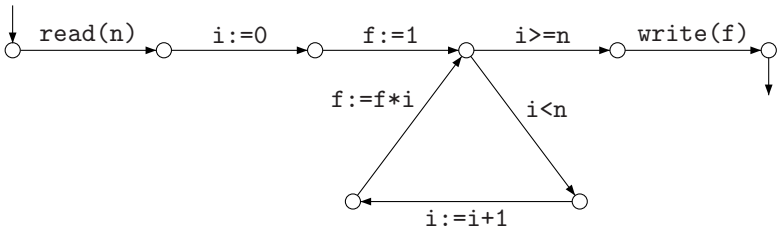

Figure 2.8: PASCAL program.

(a) all strings with exactly one a;

(b) all strings with at least one a;

(c) all strings with at most one a;

(d) all strings with no more than three a's;

(e) all strings with at least two a's and at most two b's.

2.1.2 Let $\mathcal{A} = \{a, b\}$. Construct an automaton for each of the following languages:

(a) The language of all strings in which each occurrence of a is immediately followed by an occurrence of b;

(b) The language of all strings in which each maximal a-substring has even length;

(c) The language of all strings in which both aa and bb do not occur as substrings;

(d) The language of all strings in which the number of a's is even and the number of b's is odd;

(e) The language of all strings in which neither ab nor bb occur as a substring;

(f) The language of all strings where the number of a's and twice the number of b's is divisible by 5;

(g) $L = \{a^n b a^m \mid n \equiv_3 m\}$, i.e., the language of all strings of the form $a^n b a^m$ where n and m are equal modulo 3.

2.1.3 Give an automaton for the following languages:

(a) $L = \{abwb \mid w \in \{a, b\}^*\}$;

(b) $L = \{ab^n a^m \mid n \geq 2, m \geq 3\}$;

(c) $L = \{uabv \mid u, v \in \{a, b\}^*\}$.

2.1.4 Find an automaton for the following languages, $\mathcal{A} = \{a, b\}$:

(a) $L = \{w \mid |w| \bmod 3 = 0\}$;

(b) $L = \{w \mid \#_a(w) \bmod 3 = 0\}$:

(c) $L = \{w \mid \#_a(w) - \#_b(w) > 1\}$:

2.1.5 Give an automaton for the language of all strings over $\{a, b, c\}$ where the first a precedes the first b or the first b precedes the first a.

2.1.6 Consider the set of all strings on $\{0, 1\}$ defined by the requirements below. For each, construct an accepting automaton.

(a) Every 00 is immediately followed by a 1. Thus, strings 0001 or 00100 are not in the set.

(b) All strings containing 00 but not 000.

(c) All strings of which the leftmost symbol differs from the rightmost symbol.

(d) All strings that contain two consecutive identical symbols, but never three consecutive identical symbols.

2.1.7 Give an automaton that accepts the language that contains all strings representing floating point numbers. Assume the following syntax for floating point numbers. A floating point number is an optional sign, followed by a decimal number followed by an optional exponent. A decimal number may be of the form x or $x.y$ where x and y are nonempty strings of decimal digits. An exponent begins with E and is followed by an optional sign and then an integer. An integer is a nonempty string of decimal digits. Make sure that any superfluous leading 0's are not allowed. E.g., the string 007 should not be allowed.

2.1.8 Give an automaton that accepts all strings over the alphabet $\{a, b, c, \ldots, x, y, z\}$ in which the vowels a, e, i, o, and u, occur in alphabetical order. The language should thus accept strings *abstemious*, *facetious*, *sacreligious*, and *apple*. But it does not contain *tenacious*.

2.1.9 Give an automaton that accepts all strings over the alphabet $\{a, b, c, d\}$ such that at least one of the symbols of this alfabet does not occur in the string.

2.1.10 For each of the statements below, decide whether it is true or false. If it is true, prove it. If not, give a counterexample. All parts refer to languages over the alphabet $\{a, b\}$.

(a) If $L_1 \subseteq L_2$ and L_1 is not regular, then L_2 is not regular.

(b) If L_1 is regular, L_2 is not regular, and $L_1 \cap L_2$ is regular, then $L_1 \cup L_2$ is not regular.

2.1.11 Suppose that a certain programming language permits only identifiers that begin with a letter, contain at least one but no more than three digits, and can have any number of letters. Give an automaton that accepts precisely all such identifiers.

2.1.12 In the roman number system, numbers are represented by strings over the alphabet $\{M, D, C, L, X, V, I\}$. Give an automaton that accepts such strings only if they are properly formed roman numbers. For simplicity, replace the "subtraction" convention in which the number nine is represented by IX with an additional equivalent that uses $VIIII$ instead.

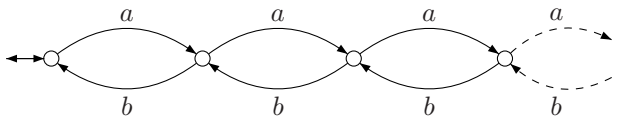

Figure 2.9: Counter.

2.1.13 Construct an automaton that accepts a string over $\{0, 1\}$ if and only if the value of the string, interpreted as a binary representation of an integer, is divisible by 5.

2.1.14 Let us define an operation *truncate*, which removes the rightmost symbol from any string. The operation can be extended to languages by

$$\mathit{truncate}(L) = \{\mathit{truncate}(w) \mid w \in L\}.$$

Prove that, in case L is regular, also $\mathit{truncate}(L)$ is regular.

2.1.15 Design an automaton with no more than five states that accepts the language $\{abab^n \mid n \geq 0\} \cup \{aba^n \mid n \geq 0\}$.

2.1.16 Find an automaton with three states that accepts the language $\{a^n \mid n \geq 1\} \cup \{b^m a^k \mid m, k \geq 0\}$. Can this be done with an automaton with fewer states?

2.1.17 Show that the language $L = \{vwv \mid v, w \in \{a, b\}^*, |v| = 1\}$ is regular.

2.1.18 Show that the language $L = \{a^n \mid n \geq 0, n \neq 3\}$ is regular.

2.1.19 Show that the language $L = \{a^n \mid n$ is a multiple of 3, but not a multiple of 5 $\}$ is regular.

2.1.20 Suppose the language L is regular. Show that $L - \{\varepsilon\}$ is regular.

2.1.21 Suppose the language L is regular and $a \in \mathcal{A}$. Show that $L \cup \{a\}$ is regular.

2.1.22 Suppose M is an automaton with states $s, t \in \mathcal{S}$ such that $s \stackrel{vw}{\longrightarrow\!\!\!\!\!\rightarrow} t$. Show that there is a state $u \in \mathcal{S}$ with $s \stackrel{v}{\longrightarrow\!\!\!\!\!\rightarrow} u$ and $u \stackrel{w}{\longrightarrow\!\!\!\!\!\rightarrow} t$.

2.1.23 Suppose we change the definition of an automaton so that more than one initial state is allowed. Give this definition formally. A string w is accepted by such an automaton if from every initial state, there is a path with label w to a final state. Show that every language accepted by such an automaton with multiple initial states is regular.

2.1.24 Let the language L over alphabet $\mathcal{A} = \{a, b\}$ be defined as follows: $L = \{w \in \mathcal{A}^* \mid \#_a(w) = \#_b(w)$ and for all prefixes v of w we have $\#_a(v) \geq \#_b(v)\}$. This language is sometimes called the *bracket* language (to see this, interpret a as open bracket, and b as close bracket). Argue that this language is accepted by the transition system shown in Figure 2.9. This is a counter, interpreting a as an increment and b as a decrement. Show as in the proof of Theorem 2.8, that this language is not regular.

2.2 Automata and processes

In the previous section we looked at an automaton as generating a set of strings, a language. But widely different automata can generate the same language. Just looking at an automaton as an input-output function, considering the language is sufficient, but when we consider the model of interacting automata, more can be observed of an automaton.

The basic observations of an automaton are the execution of an action (a step $. \xrightarrow{a} .$) and a termination ($. \downarrow$). Automata that have the same observations can be considered to be the same. But we have to take care in formalising this.

First of all, we already stated we do not find the names of states to be relevant. States cannot be observed directly, they can only be distinguished by the steps they do or do not allow.

Second, all the reachable states of an automaton are relevant, not just the states that can lead to a final state. Also steps that lead to a deadlock can be observed.

Third, two states that have the same observations can be identified. If there are two states that only allow a step a followed by a step b followed by termination $\downarrow$, then there is no observation that can tell the two states apart.

The fourth point is the most subtle one. It states that the moment of choice is relevant. We illustrate with an example.

Example 2.13. We have a look at Frank Stockton's story "The Lady or the Tiger?" (see [15]).

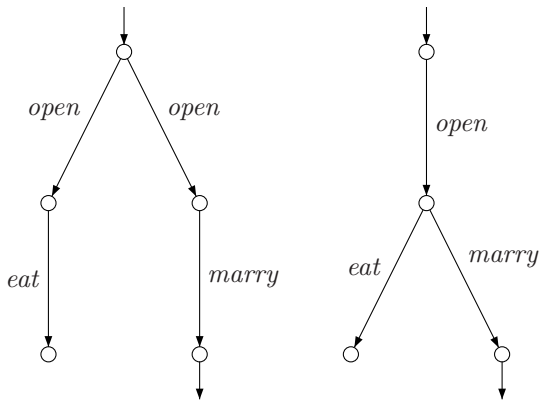

Figure 2.10: The lady or the tiger?

A prisoner is confronted with two closed doors. Behind one of the doors is a dangerous tiger, and behind the other there is a beautiful lady. If the prisoner opens the door hiding the tiger he will be eaten. On the other hand, if he opens the door hiding the beautiful lady, the lady and he will get married and he will be free. Unfortunately, the prisoner does not know what hides behind what door.

This situation can be described as an automaton depicted in Figure 2.10 on the left-hand side, using the following actions:

1. *open* representing the act of opening a door;

2. *eat* representing the act of (the tiger) eating the young man;

3. *marry* representing the act of the beautiful lady and the prisoner getting married.

This automaton models the fact that after a door has been opened the prisoner is confronted with either the tiger or the beautiful lady. He does not have the possibility to select his favorite, which conforms to the description above. Note that step *marry* ends in a final state, whereas step *eat* does not. This can be interpreted to mean that the *marry* step results in a *successful* termination of the process, and that the *eat* transition results in *unsuccessful* termination. A non-terminating state in which no actions are possible, such as the state resulting from the *eat* transition, is often called a *deadlock* state.

Now have a look at the right-hand automaton in Figure 2.10. As the left-hand automaton, it has the language $\{open.marry\}$. However, there are good reasons why the situation modeled by the right automaton of Figure 2.10 is different from the situation described by the other automaton. In the right automaton, the choice between the tiger and the lady is only made after opening a door. It could describe a situation with only one door leading to both the tiger and the lady; in this case, the prisoner still has a choice after opening a door. Clearly, this situation differs considerably from the situation described above.

The set of observations of the automata differs: in the left-hand automaton, a step *open* can be observed leading to a state where only the observation *eat* is possible, and a step *open* can be observed leading to a state where only the observation *marry* is possible. On the right-hand side, there is only the observation of step *open* to a state where both observations *eat* and *marry* are possible. Thus, the middle state on the right differs from both middle states on the left, as its set of observations is different. Thus, if for some reason action *eat* is impossible, is blocked, then the automaton on the left can become stuck after the execution of an *open* action, whereas the one on the right cannot. The choice whether or not to execute *eat* or *marry* in the left automaton is made (implicitly) upon execution of the *open* action in the initial state, whereas the same choice is made only after execution of the initial *open* action in the right automaton. It is said that the automata have different *branching structure*.

It is often desirable to be able to distinguish between automata with the same strings of actions that have different termination behavior or that have different branching structure. In order to do this, a notion of equivalence is defined that is finer than language equivalence, in the sense that it distinguishes automata accepting the same language but with different termination behavior or branching structure. We will focus on bisimulation equivalence. However, there are many other equivalence relations in between language equivalence and bisimulation equivalence, that could also serve the purpose. In literature, there is a lengthy debate going on as to which equivalence is the 'right' one to use in this situation. We will not enter this debate, and just note that bisimulation is the one most used and most studied, and bisimulation will certainly meet all the requirements that we might come up with.

Exercises

2.2.1 Give an automaton for a traffic light. The actions are the colours of the traffic light: *red*, *yellow*, *green*. Initially, the traffic light is red.

2.2.2 Give an automaton for a crossing of two one-way streets, where each street has one traffic light. The actions involved are red_1, $yellow_1$, $green_1$, red_2, $yellow_2$, $green_2$. Make sure that collisions cannot occur, assuming that drivers respect the traffic lights. Initially, both traffic lights are red.

2.2.3 Give an automaton for an elevator. The elevator serves 5 floors numbered 0 through 4, starting at floor 0. The actions are *up*, *down* denoting a move to the next floor up resp. down.

2.2.4 Consider a stopwatch with two buttons and one display. Initially, the display is empty. As soon as the start button is pressed, the stopwatch starts counting time in seconds from zero up. Pushing the stop button results in stopping the counting of seconds. After stopping the counting, on the display the amount of time that has elapsed is displayed. Use actions *start*, *stop*, *tick*, *display*(s) (s a natural number). Draw a transition system. What happens when the start button is pushed while counting? Can termination occur?

2.3 Bisimulation

We established that language equivalence is not sufficient to capture the behaviour of an interacting automaton. In the previous section, we argued that it is important to observe the difference of states that have a different set of outgoing actions: we can see the difference between a state that has both an outgoing a- and b-action, and a state that has just one of the two. Thus, we can see whether or not a state exists with a certain set of outgoing actions, but not how many times such a state occurs. The notion of bisimulation does exactly this: related states have the same set of outgoing actions, but duplication of states is disregarded.

Definition 2.14. Let $M = (\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \downarrow)$ and $M' = (\mathcal{S}', \mathcal{A}, \rightarrow', \uparrow', \downarrow')$ be two automata with the same alphabet. The automata M and M' are *bisimilar*, $M \leftrightarrows M'$, iff there is a relation R between their reachable states that preserves transitions and termination, i.e.

1. R relates reachable states: every reachable state of M is related to a reachable state of M' and every reachable state of M' is related to a reachable state of M;

2. $\uparrow$ is related by R to $\uparrow'$;

3. whenever s is related to s', sRs' and $s \xrightarrow{a} t$, then there is state t' in M' with $s' \xrightarrow{a}{}' t'$ and tRt'; this is the *transfer condition* from left to right (see Figure 2.11);

4. whenever s is related to s', sRs' and $s' \xrightarrow{a}{}' t'$, then there is state t in M with $s \xrightarrow{a} t$ and tRt'; this is the *transfer condition* from right to left (see Figure 2.12);

5. whenever sRs', then $s \downarrow$ if and only if $s' \downarrow'$ (see Figures 2.13 and 2.14).

The relation R links states in the two automata that have the same behavior.

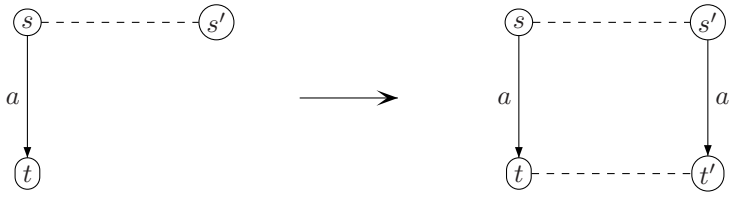

Figure 2.11: Transfer condition from left to right.

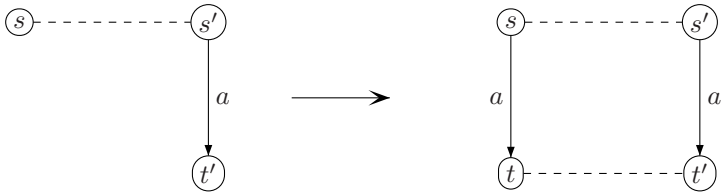

Figure 2.12: Transfer condition from right to left.

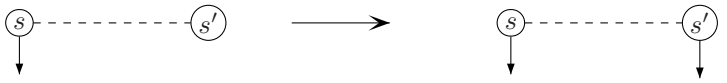

Figure 2.13: Transfer condition for final states from left to right.

One can think of the notion of bisimilarity in terms of a two-person game. Suppose two players each have their own automaton. The game is played as follows. First, one of the players makes a transition or move from the initial state. The role of the other player is to match this move precisely, also starting from the initial state. Next, again one of the players makes a move. This does

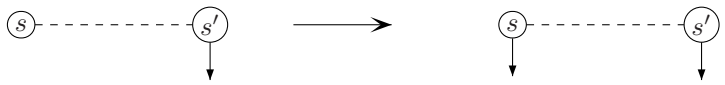

Figure 2.14: Transfer condition for final states from right to left.

not have to be the same player as the one that made the first move. The other must try to match this move, and so on. If both players can play in such a way that at each point in the game any move by one of the players can be matched by a move of the other player, then the automata are bisimilar. Otherwise, they are not.

Example 2.15. In Figure 2.15, an example of a bisimulation relation on automata is given. Related states are connected by a dashed line.

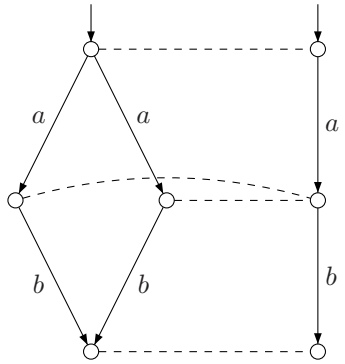

Figure 2.15: Example of bisimulation.

Example 2.16. Figure 2.16 recalls two by now well-known automata. It should not come as a surprise that they are not bisimilar. States that can possibly be related are connected by a dashed line. The states where the behaviors of the two systems differ are indicated by dashed lines labeled with a question mark. None of the two indicated pairs of states satisfies the transfer conditions of Definition 2.14.

So far, bisimilarity is just a relation on transition systems. However, it has already been mentioned that it is meant to serve as a notion of equality. For that purpose, it is necessary that bisimilarity is an equivalence relation. It is not difficult to show that bisimilarity is indeed an equivalence relation.

Theorem 2.17 (Equivalence)**.** Bisimilarity is an equivalence.

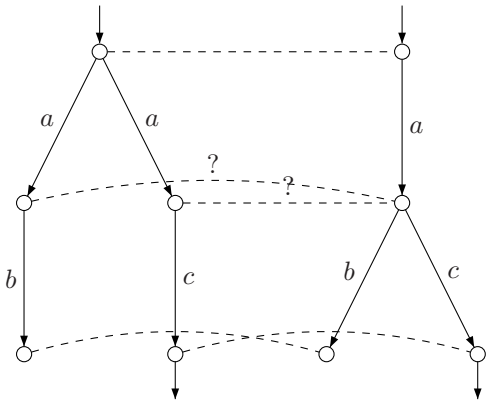

Figure 2.16: Two automata that are not bisimilar.

Proof. Proving that a relation is an equivalence means that it must be shown that it is reflexive, symmetric, and transitive. Let $M = (\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \downarrow)$ be an automaton. First, it is not hard to see that the relation $R = \{(s, s) \mid s \in \mathcal{S}, \uparrow \longrightarrow\!\!\!\!\!\rightarrow s\}$ is a bisimulation relation. This implies that $M \leftrightarrows M$.

Second, assume that $M \leftrightarrows M'$ for automata M, M'. If R is a bisimulation relation relating the reachable states of M and M', then the relation $R' = \{(t, s) \mid sRt\}$ is a bisimulation relation as well, and sRt if and only if $tR's$. Hence, $M' \leftrightarrows M$, proving symmetry of $\leftrightarrows$.

Finally, for transitivity of $\leftrightarrows$, it must be shown that the relation composition of two bisimulation relations results in a bisimulation relation again. Let M, M', M'' be automata, and let R_1 and R_2 be bisimulation relations between the reachable states of M and M', respectively M' and M''. The relation composition $R_1 \circ R_2$, defined by $sR_1 \circ R_2t$ iff there is a u with sR_1u and uR_2t, is a bisimulation relating the reachable states of M and M'', implying transitivity of $\leftrightarrows$. □

Two automata that are bisimilar are certainly language equivalent.

Theorem 2.18. If $M \leftrightarrows M'$, then $M \approx M'$.

Proof. Take a string $w \in \mathcal{L}(M)$. By definition, this means there is a path $\uparrow \xrightarrow{w}\!\!\!\!\!\rightarrow s$ to a state s in M with $s \downarrow$. Take a bisimulation relation R between M and M'. For each state in the given path, we can find a state in M' that is bisimilar to it, and is connected by the same step. In particular, R relates $\uparrow$ to $\uparrow'$ and s to some state s' in M' with $\uparrow' \xrightarrow{w}\!\!\!\!\!\rightarrow' s'$ and $s' \downarrow'$. This means $w \in \mathcal{L}(M')$. The other direction is similar. □

Obviously, the reverse of this theorem does not hold. This means that the set of automata is divided into a set of equivalence classes of automata that are language equivalent, and that each of these language equivalence classes is divided into a number of bisimulation equivalence classes. A bisimulation equivalence class of automata is called a *regular process*.

We can use the definition of bisimulation just as well on (infinite) transition systems. Consider again the transition system of the simple counter presented earlier (2.6), reproduced here.

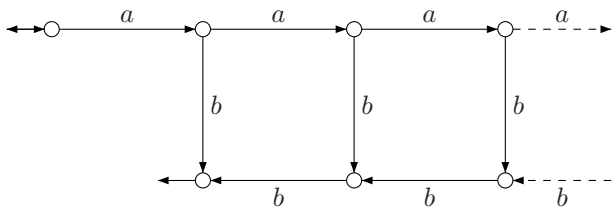

Figure 2.17: Simple counter.

Notice that no two states of this transition system can be related by a bisimulation relation (every state on the top row is uniquely characterized by the number of consecutive b-steps that can be executed, together with the possibility to execute an a-step, every state on the bottom row also has a unique number of consecutive b-steps possible, and no possibility to execute a). Thus, this transition system is not bisimilar to any finite automaton, the system does not denote a regular process.

Definition 2.19. A *regular process* is a bisimulation equivalence class of transition systems that contains a finite automaton. We say that a transition system that is bisimilar to a finite automaton *denotes a regular process.*

It can also be seen from Theorem 2.8 that the transition system of Figures 2.17 and 2.6 does not denote a regular process: since the language of this transition system is not regular, it cannot be language equivalent to a finite automaton, and thus it cannot be bisimilar to a finite automaton.

Another characterization of bisimulation is in terms of *coloring.*

Definition 2.20. Let C be a given finite set, the set of *colors.* A *coloring* of an automaton $M = (\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \downarrow)$ is a mapping c from the reachable states of M to C. Given a coloring, and a path $s \stackrel{w}{\longrightarrow\!\!\!\!\!\rightarrow} t$ from s to t with label w, say $w = a_0 a_1 \ldots a_n$, the *colored path* from s to t is the alternating sequence of steps and colors $c(s) \stackrel{a_0}{\longrightarrow} c(..) \stackrel{a_1}{\longrightarrow} \ldots \stackrel{a_n}{\longrightarrow} c(t)$. We can also denote that the endpoint of a path is final, in this case also $c(s) \stackrel{a_0}{\longrightarrow} c(..) \stackrel{a_1}{\longrightarrow} \ldots \stackrel{a_n}{\longrightarrow} c(t) \downarrow$ is a colored path. We call a coloring *consistent* if every two states that have the same color have the same colored paths starting from them.

Theorem 2.21. Any consistent coloring on automaton M is a bisimulation between M and M (an *auto*bisimulation on M). Any autobisimulation on M gives a consistent coloring, by coloring any related states the same.

We can also color any pair of automata. Then, two automata are bisimilar exactly when there is a consistent coloring on both automata that gives the initial states the same color.

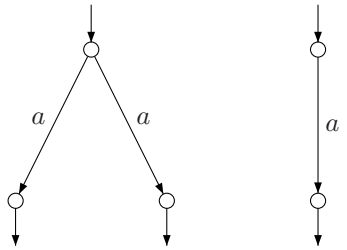

Figure 2.18: Bisimilar automata?

Theorem 2.22. Let R and R' be two bisimulations between M and M'. Then also $R \cup R'$ is a bisimulation between M and M'.

Thus, the union of bisimulations is again a bisimulation. If two automata are bisimilar, then the *maximal bisimulation* between the two automata is obtained by taking the union of all bisimulations between them.

Given an automaton, we can reduce the automaton as much as possible by identifying all states that are related by the maximal autobisimulation. This will give the smallest automaton that is bisimilar to the given automaton. In it, a consistent coloring will color all states differently. This smallest automaton is often used as the representative of its bisimulation equivalence class.

To end this section, we consider deadlock.

Definition 2.23 (Deadlock)**.** A state s of an automaton is a *deadlock state* if and only if it does not have any outgoing transitions and it does not allow successful termination, i.e., if and only if for all $a \in \mathcal{A}$, $s \stackrel{a}{\not\longrightarrow}$, and $s \notin \downarrow$. A transition system *has a deadlock* if and only if it has a reachable deadlock state; it is *deadlock free* if and only if it does not have a deadlock.

An important property that has already been suggested when motivating the notion of bisimilarity is that bisimilarity preserves deadlocks. Exercise 4 asks for a proof of this fact.

Exercises

2.3.1 Are the pairs of automata in Figures 2.18, 2.19, 2.20, 2.21, 2.22 bisimilar? If so, give a bisimulation between the two automata; otherwise, explain why they are not bisimilar.

2.3.2 Give a bisimulation between any two of the automata in Figure 2.23.

2.3.3 Which of the transition systems of Exercises 1 and 2 are deadlock free?

2.3.4 Let M and M' be two *bisimilar* automata. Show that M has a deadlock if and only if M' has a deadlock.

2.3.5 Consider the two automata in Figure 2.24. If they are bisimilar, exhibit a bisimulation. If they are not bisimilar, explain why not. For each

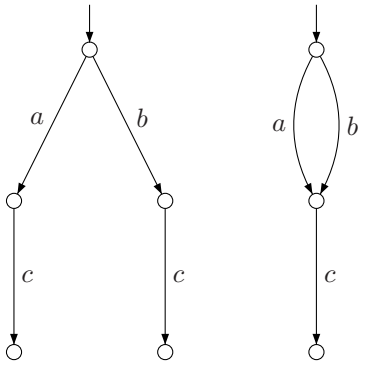

Figure 2.19: Bisimilar automata?

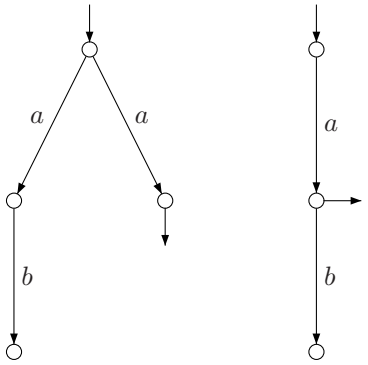

Figure 2.20: Bisimilar automata?

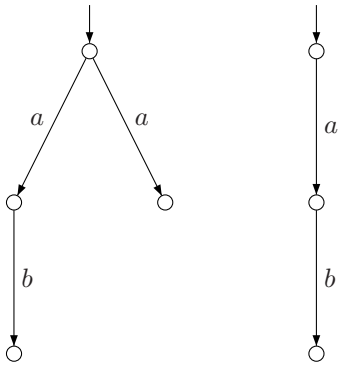

Figure 2.21: Bisimilar automata?

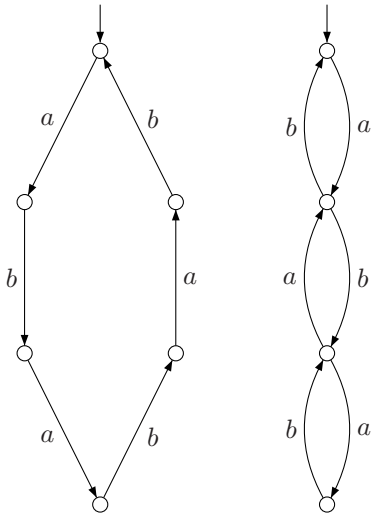

Figure 2.22: Bisimilar automata?

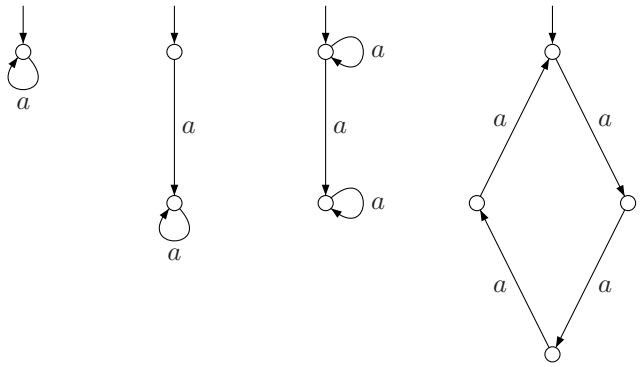

Figure 2.23: All automata are bisimilar.

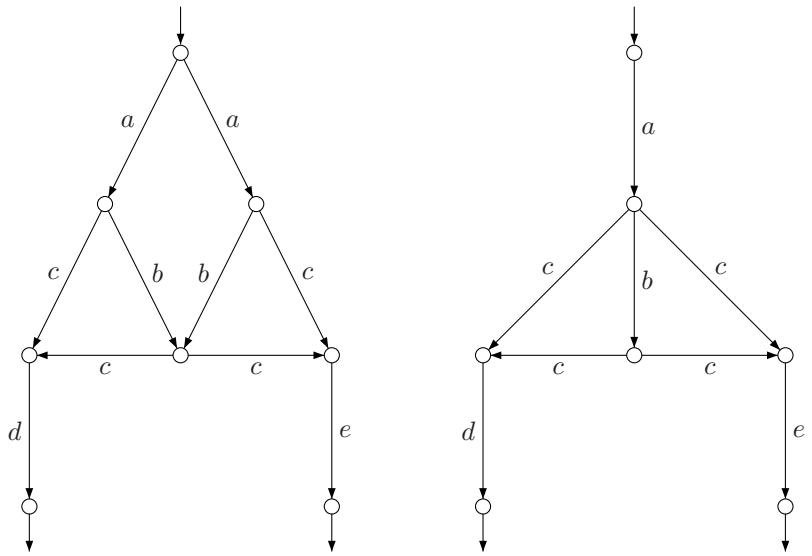

Figure 2.24: Bisimilar automata?

automaton, reduce it as much as possible with respect to bisimulation. Next, do the exercise again with e replaced by d.

2.3.6 Prove that the relation composition of two bisimulation relations is again a bisimulation.

2.3.7 Prove that the transition system of Exercise 24 in Section 2.1 does not denote a regular process.

2.4 Recursive specifications

While the visualization of an automaton is intuitively appealing and useful in many respects, it is not so useful mathematically, for doing calculations. For this reason, we will consider a presentation of automata as mathematical expressions, in an algebra.

Definition 2.24. We proceed to define the Minimal Algebra MA. MA has a very simple syntax:

1. There is a constant **0**; this denotes inaction, no possibility to proceed, a deadlock state;
2. There is a constant **1**; this denotes termination, a final state;
3. For each element of the alphabet $\mathcal{A}$, there is a unary operator $a.\_$ called *action prefix*; a term $a.x$ will execute the elementary action a and then proceed as x;
4. There is a binary operator $+$ called *alternative composition*; a term $x + y$ will either execute x or execute y, a choice will be made between the alternatives.

Thus, MA has expressions $a.\mathbf{0}, (a.\mathbf{0}) + (a.\mathbf{1}), a.((b.\mathbf{1}) + \mathbf{0})$. We use brackets and sometimes omit the symbol of action prefixing as is customary in arithmetic, so e.g. $ab\mathbf{0} + cd\mathbf{1} + e\mathbf{0}$ denotes $((a.(b.\mathbf{0})) + (c.(d.\mathbf{1}))) + (e.\mathbf{0})$.

Every term over this syntax will denote an automaton. Before defining this automaton, we first interpret every term as a state in an automaton, and define when such a state is a final state, and when such a state has a transition to another state. We do this by a method which is called *Structural Operational Semantics*, SOS for short. It defines the basic observations of a term.

Definition 2.25. 1. We have $\mathbf{1}\downarrow$, a state named $\mathbf{1}$ is a final state;

2. For all $a \in \mathcal{A}$ and all terms x, $a.x \xrightarrow{a} x$, a state named $a.x$ has a transition labeled a to state x;

3. For all terms x, y, z and all $a \in \mathcal{A}$ we have that $x + y \xrightarrow{a} z$ whenever $x \xrightarrow{a} z$ or $y \xrightarrow{a} z$; a state $x + y$ has a transition to a certain state exactly when x or y has such a transition;

4. For all terms x, y we have that $x + y \downarrow$ whenever $x \downarrow$ or $y \downarrow$; $x + y$ is a final state exactly when x or y is a final state.

Notice that nothing is defined for term $\mathbf{0}$. This means $\mathbf{0}$ is not a final state, and has no outgoing transition. Moreover, notice that when a term y is reachable from x, $x \twoheadrightarrow y$, then y is a subterm of x. Thus, from a given term, only finitely many terms are reachable. For future reference, we list the rules of Definition 2.25 in Table 1.

$$\frac{}{\mathbf{1}\downarrow} \qquad \frac{}{a.x \xrightarrow{a} x}$$

$$\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'} \qquad \frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'} \qquad \frac{x \downarrow}{x + y \downarrow} \qquad \frac{y \downarrow}{x + y \downarrow}$$

Table 1: Operational rules for MA ($a \in \mathcal{A}$).

We use a particular format to present these rules. Each rule has a central dividing line. Above the line are the *premises*, below the *conclusion*. If a rule has no premises, it is called an *axiom*.

Definition 2.26. We define the automaton of term t, $\mathcal{M}(t)$, as follows:

1. The set of states is the set of terms reachable from t;

2. The alphabet is $\mathcal{A}$;

3. The initial state is t;

4. The set of transitions and the final states are defined by means of the operational rules in Table 1.

The language of a term t, $\mathcal{L}(t)$, is just $\mathcal{L}(\mathcal{M}(t))$.

Figure 2.25: Automata of $\mathbf{0}, \mathbf{1}, a.\mathbf{1}$.

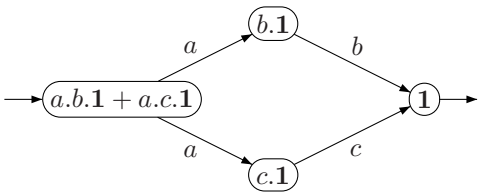

Figure 2.26: Automaton of $a.b.\mathbf{1} + a.c.\mathbf{1}$.

Figure 2.25 shows the automata of terms $\mathbf{0}, \mathbf{1}$ and $a.\mathbf{1}$.

A more complicated example is given in Figure 2.26. Since $a.b.\mathbf{1} \xrightarrow{a} b.\mathbf{1}$, the initial state has an a-labeled transition to $b.\mathbf{1}$. The other transition from the initial state follows from $a.c.\mathbf{1} \xrightarrow{a} c.\mathbf{1}$.

We can calculate the languages accepted by the terms over this syntax.

Theorem 2.27. The languages accepted by terms over MA are as follows:

1. $\mathcal{L}(\mathbf{0}) = \emptyset$;
2. $\mathcal{L}(\mathbf{1}) = \{\varepsilon\}$;
3. $\mathcal{L}(a.x) = \{aw \in \mathcal{A}^* \mid w \in \mathcal{L}(x)\}$;
4. $\mathcal{L}(x + y) = \mathcal{L}(x) \cup \mathcal{L}(y)$.

Proof.

1. See Figure 2.25. $\mathcal{M}(\mathbf{0})$ has no final state.
2. See Figure 2.25. The only path from the initial state to itself has label ε.
3. Suppose there is a path from the initial state to a final state with label w. As the initial state is not final (there is no way to derive $a.x \downarrow$), and the only step that can be taken from the initial state has label a, we must have $w = av$ for some string v. Now v must be the label of a path from the initial state of $\mathcal{M}(x)$ to a final state, so $v \in \mathcal{L}(x)$.
4. Suppose $w \in \mathcal{L}(x + y)$. If $w = \varepsilon$, then $(x + y) \downarrow$, so either $x \downarrow$ or $y \downarrow$ and $w \in \mathcal{L}(x)$ or $w \in \mathcal{L}(y)$. Otherwise, there are $a \in \mathcal{A}$ and a string v with $w = av$. Then $x + y \xrightarrow{a} p$, so either $x \xrightarrow{a} p$ or $y \xrightarrow{a} p$. Again we obtain $w \in \mathcal{L}(x)$ or $w \in \mathcal{L}(y)$.

□

Notice the difference between $\mathcal{L}(\mathbf{0})$ and $\mathcal{L}(\mathbf{1})$: the latter accepts the empty string, the former accepts no string.

We considered the language of terms in MA, now let us consider bisimulation. From the definition of an automaton of a term we derive the following principles of calculation.

Theorem 2.28. The following laws hold for MA terms:

1. $x + y \leftrightarrows y + x$,
2. $(x + y) + z \leftrightarrows x + (y + z)$,
3. $x + x \leftrightarrows x$,
4. $x + \mathbf{0} \leftrightarrows x$.

Proof. 1. $x + y \leftrightarrows y + x$, the order of the summands does not matter. The automaton of $x + y$ is exactly the same as the automaton of $y + x$, apart from the name of the initial state. Thus, the identity relation on states together with the pair $(x + y, y + x)$ is a bisimulation relation. For, if $x + y \xrightarrow{a} z$ for certain $a \in \mathcal{A}$ and term z, then we must have $x \xrightarrow{a} z$ or $y \xrightarrow{a} z$. But then also $y + x \xrightarrow{a} z$. Similarly, if $x + y \downarrow$, then $x \downarrow$ or $y \downarrow$. But then also $y + x \downarrow$.

2. $(x+y)+z \leftrightarrows x+(y+z)$, in case there are more than two summands it does not matter how they are grouped. The automaton of $(x+y)+z$ is exactly the same as the automaton of $x + (y + z)$, apart from the name of the initial state. Thus, the identity relation on states together with the pair $((x+y)+z, x+(y+z))$ is a bisimulation relation. For, if $(x+y)+z \xrightarrow{a} p$ for certain $a \in \mathcal{A}$ and term p, then we must have $x + y \xrightarrow{a} p$ or $z \xrightarrow{a} p$. The former implies $x \xrightarrow{a} p$ or $y \xrightarrow{a} p$. If $y \xrightarrow{a} p$ or $z \xrightarrow{a} p$, then also $y + z \xrightarrow{a} p$. Otherwise $x \xrightarrow{a} p$, so in both cases $x + (y + z) \xrightarrow{a} p$. Similarly $(x + y) + z \downarrow$ just in case $x + (y + z) \downarrow$.

3. $x+x \leftrightarrows x$, duplicate alternatives can be removed. The automaton of $x+x$ is exactly the same as the automaton of x, apart from the name of the initial state. Thus, the identity relation on states together with the pair $(x + x, x)$ is a bisimulation relation. For, if $x + x \xrightarrow{a} y$ we must have $x \xrightarrow{a} y$, and if $x \xrightarrow{a} y$ then also $x + x \xrightarrow{a} y$. Likewise $x + x \downarrow$ just in case $x \downarrow$.

4. $x + \mathbf{0} \leftrightarrows x$, there is only a deadlock state if there are no alternatives. The automaton of $x + \mathbf{0}$ is exactly the same as the automaton of x, apart from the name of the initial state. Thus, the identity relation on states together with the pair $(x + \mathbf{0}, x)$ is a bisimulation relation. For, if $x + \mathbf{0} \xrightarrow{a} y$ then we must have $x \xrightarrow{a} y$, as $\mathbf{0} \xrightarrow{a} y$ cannot occur. Likewise $x + \mathbf{0} \downarrow$ exactly when $x \downarrow$.

□

We conclude that the + operator on automata is commutative, associative, idempotent, and has $\mathbf{0}$ as a unit element. For future reference, we list these laws in Table 2. As bisimilarity implies language equivalence, these laws also hold for $\approx$.

Apart from the laws shown in Table 2, MA has two more laws, the distributivity of action prefix over alternative composition and the law for zero shown

$$\begin{array}{ll} x + y & \leftrightarrows y + x \\ (x + y) + z & \leftrightarrows x + (y + z) \\ x + x & \leftrightarrows x \\ x + \mathbf{0} & \leftrightarrows x \end{array}$$

Table 2: Bisimulation laws of MA.

$$\begin{array}{ll} a.x + a.y & \approx a.(x + y) \\ a.\mathbf{0} & \approx \mathbf{0} \end{array}$$

Table 3: Language equivalence laws of MA ($a \in \mathcal{A}$).

in Table 3. Application of these laws will change an automaton to a different automaton, but will preserve language acceptance.

Theorem 2.29. Let x, y be two terms over MA. Then automata $a.(x + y)$ and $a.x + a.y$ accept the same language.

Proof. $\mathcal{L}(a.(x + y)) = \{aw \mid w \in \mathcal{L}(x + y)\} = \{aw \mid w \in \mathcal{L}(x) \cup \mathcal{L}(y)\} = \{aw \mid w \in \mathcal{L}(x)\} \cup \{aw \mid w \in \mathcal{L}(y)\} = \mathcal{L}(a.x) \cup \mathcal{L}(a.y) = \mathcal{L}(a.x + a.y)$. □

Theorem 2.30. Automata $a.\mathbf{0}$ and $\mathbf{0}$ accept the same language.

Proof. $\mathcal{L}(a.\mathbf{0}) = \mathcal{L}(\mathbf{0}) = \emptyset$. □

The distributive law will turn out to be very useful in the following. For the application of the laws, we will use the fact that language equivalence is an equivalence relation, see the previous section. Furthermore, we also need to be able to apply the laws in context, i.e. we need that language equivalence is a congruence relation. By this, we mean the following.

Theorem 2.31. Let x, y be two terms over MA, and suppose the automata denoted by x and y accept the same language, $x \approx y$. Then:

1. For all actions $a \in \mathcal{A}$, also $a.x$ and $a.y$ accept the same language, $a.x \approx a.y$;
2. For all terms z, also $x+z$ and $y+z$ accept the same language, $x+z \approx y+z$.

Proof. Straightforward. □

Also bisimilarity is an equivalence relation. It is also a congruence relation.

Theorem 2.32. Bisimilarity is a congruence relation on MA terms.

Proof.
1. If $x \leftrightarrows y$ and $a \in \mathcal{A}$, then $a.x \leftrightarrows a.y$. If R is a bisimulation relation between $\mathcal{M}(x)$ and $\mathcal{M}(y)$, then $R' = R \cup \{(a.x, a.y)\}$ is a bisimulation relation between $\mathcal{M}(a.x)$ and $\mathcal{M}(a.y)$;
2. If $x \leftrightarrows y$ and $x' \leftrightarrows y'$, then $x + x' \leftrightarrows y + y'$. Take a bisimulation relation R relating $\mathcal{M}(x)$ and $\mathcal{M}(y)$ and a bisimulation relation R' relating $\mathcal{M}(x')$ and $\mathcal{M}(y')$. Then $R'' = R \cup R' \cup \{(x+x', y+y')\}$ is a bisimulation relation between $\mathcal{M}(x + x')$ and $\mathcal{M}(y + y')$. It is in taking this union that one state can become related to more than one state.

□

Thus, both language equivalence and bisimilarity are congruence relations on MA. This means we can do calculations with both $\approx$ and $\leftrightarrows$. In the former case, more laws can be used.

It is obvious that not all automata can be represented by a term over MA, as loops or edges back to a previous state cannot occur: in a path the terms of the states traversed can only become smaller. So, if we want to represent any given automaton, extra terms are needed. We do this by adding an extra name to the syntax for every state of the automaton, and defining these names by the steps they can take.

As an example, look at the automaton in Figure 2.27. This automaton has states S, T, U, V, W, R. S is the initial state (we often use the letter S for the *start* state).

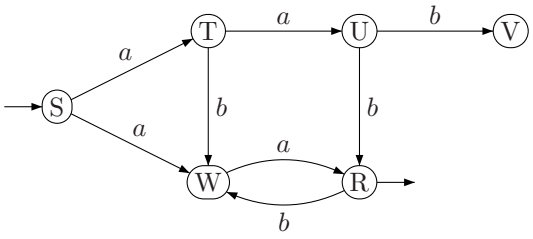

Figure 2.27: Example automaton.

Then we associate the following set of equations:

$$\begin{aligned} S &= a.T + a.W \\ T &= a.U + b.W \\ U &= b.V + b.R \\ V &= \mathbf{0} \\ W &= a.R \\ R &= \mathbf{1} + b.W. \end{aligned}$$

In these equations, we use the = symbol for equality. We will find out that the two sides of each equation yield bisimilar automata.

The presentation of an automaton as a set of recursive equations is called a *recursive specification* or a *grammar*. For each name added, there is exactly one equation, the defining equation of this state. But the extra names again occur on the right-hand sides of the equations, so they are *recursive* equations.

Definition 2.33. Let $\mathcal{N}$ be a finite set of *names* or *variables*. A *recursive specification* over $\mathcal{N}$ with initial variable $S \in \mathcal{N}$ is a set of equations of the form $P = t_P$, exactly one for each $P \in \mathcal{N}$, where each right-hand side t_P is a term over MA, possibly containing elements of $\mathcal{N}$.

A recursive specification over names $\mathcal{N}$ is called *linear* if each right-hand side t_P is a linear term. Linear terms are defined recursively:

1. terms $\mathbf{1}$, $\mathbf{0}$ or of the form $a.Q$, with $a \in \mathcal{A}$ and $Q \in \mathcal{N}$ are linear terms;

2. an alternative composition (sum) of linear terms is a linear term.

Thus, terms $a.Q + \mathbf{1} + b.R$ and $\mathbf{0} + a.Q + b.R + \mathbf{1}$ are linear. Implicitly, we use the associativity of $+$ here, as we do not write brackets in sums of more than two terms.

In a linear recursive specification, each right-hand side is an alternative composition of a number of terms each of which are an action prefix of an added name, or $\mathbf{1}$ or $\mathbf{0}$.

Given a recursive specification, we can add two extra clauses to Definition 2.25.

Definition 2.34. Suppose we have a recursive specification over names $\mathcal{N}$, so for each $P \in \mathcal{N}$, there is an equation $P = t_P$, where t_P is a term over MA with extra names $\mathcal{N}$.

Then the rules of Definition 2.25 also hold for these extended terms, and moreover we have the extra rules:

1. For $P \in \mathcal{N}$, we have $P \xrightarrow{a} x$ whenever $t_P \xrightarrow{a} x$;
2. For $P \in \mathcal{N}$, we have $P \downarrow$ whenever $t_P \downarrow$.

See Table 4.

$$\frac{t_P \xrightarrow{a} x \quad P = t_P}{P \xrightarrow{a} x} \qquad \frac{t_P \downarrow \quad P = t_P}{P \downarrow}$$

Table 4: Operational rules for recursion.

Next, we define the automaton of a recursive specification.

Definition 2.35. Suppose we have a recursive specification over names $\mathcal{N}$ and initial variable S. The *automaton* of this recursive specification, denoted $\mathcal{M}(S)$ is defined as follows:

1. The set of states is the set of terms over MA with extra names $\mathcal{N}$ that are reachable from S;
2. The alphabet is $\mathcal{A}$;
3. The initial state is S;
4. The transitions and the final states are given by the rules in Tables 4 and 1.

Also, the language of S, $\mathcal{L}(S)$, is $\mathcal{L}(\mathcal{M}(S))$.

Then, if we consider the running example, the automaton of S is again the one in Figure 2.27. Notice that if we had started with an automaton containing states not reachable from the initial state, then these would be omitted from the resulting automaton (thus, a form of *garbage collection* is incorporated in these definitions).

Now this definition can be extended to find the automaton of any term over MA containing elements of $\mathcal{N}$. For instance, if we have a recursive specification with initial variable S, then the automaton of $a.S + \mathbf{1}$ is found by prefixing the automaton of S with a a-step and marking the new initial state as a final state.

As before, we write $x \leftrightarrows y$ whenever $\mathcal{M}(x) \leftrightarrows \mathcal{M}(y)$. Also, we write $x \approx y$ whenever $\mathcal{M}(x) \approx \mathcal{M}(y)$, which means $\mathcal{L}(x) = \mathcal{L}(y)$. As before, $x \leftrightarrows y$ implies $x \approx y$, but not the other way around (to see the latter, consider e.g. $a.\mathbf{1} + b.\mathbf{0}$ and $a.\mathbf{1}$).

Theorem 2.36. Let a recursive specification contain an equation $P = t$. Then $P \leftrightarrows t$.

Proof. By the rules of Table 4, the variable P has exactly the transitions and terminations of its right-hand side. The only difference between $\mathcal{M}(P)$ and $\mathcal{M}(t)$ can be in the initial node, see the following example. □

Example 2.37. Consider the recursive specification $S = a.S$. The automaton of S is given on the left-hand side of Fig. 2.28, the automaton of $a.S$ on the right-hand side. It is easy to give a bisimulation between the two automata.

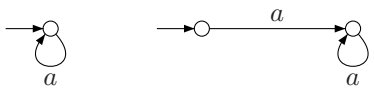

Figure 2.28: Automata of $S = a.S$.

Example 2.38. Consider the recursive specification given by equation $S = S$. According to the operational rules, S can only do a step or be a final state when S can do this step or is final. This means there is no derivation of a step or a final state, and $S \leftrightarrows \mathbf{0}$.

Similarly, whenever the equation of a variable P contains a summand P, this summand can be left out, as its presence will not add a step or final state.

The laws of Tables 2 and 3 still hold for the algebra MA extended with one or more recursive specifications. The proofs can be copied exactly, the only difference is that now the initial state might be reachable by a number of steps, as in the previous example.

Also, language equivalence and bisimilarity are still congruence relations on the extended algebra, Theorems 2.31 and 2.32 still hold. This allows us to do calculations on terms. We show this in the use of the following theorem.

Theorem 2.39. Let a recursive specification over MA and names $\mathcal{N}$ be given. This specification is bisimilar to a linear specification.

Proof. We transform each right-hand side of each equation as follows. We first look at the case where the right-hand side is just one symbol. If this symbol is $\mathbf{0}$ or $\mathbf{1}$, nothing needs to be done. If we have an equation of the form $P = Q$, with P, Q different, then replace Q in the equation of P with the right-hand side of Q (this is called expansion) and continue from there. If, after some of

these expansions, we get back a variable we have already expanded, it can be replaced by $\mathbf{0}$. If we have an equation of the form $P = P$, then also we can replace P with $\mathbf{0}$. The reason that we can do this, is that the operational rules of Table 4 will yield no step or final state for such variables.

Next, if a right-hand side is not just one symbol, then it needs to be an action prefix term or a sum term. If it is an action prefix term of the form $a.Q$ with $Q \in \mathcal{N}$, nothing needs to be done. If it is an action prefix term $a.t$ with t not a single variable, then we add a new name A to $\mathcal{N}$, replace t by A and add a new equation $A = t$. Of course, the procedure needs to be repeated with the new right-hand side t. Since in every round the right-hand side gets reduced, it will stop at some point. The remaining case is that we have a sum term. Then we look at each of the summands.

Any summand that is the variable of the equation or a variable that has already been expanded can be replaced by $\mathbf{0}$. Any other variable can be replaced by its right-hand side. Finally, any action prefix term can be dealt with by adding extra names if necessary. □

Example 2.40. We give an example of the procedure of turning a given recursive specification into a linear one. Suppose the following specification is given.

$$\begin{aligned} S &= a.(T + S) + \mathbf{0} \\ T &= \mathbf{1} + V \\ V &= a.V + T. \end{aligned}$$

By Theorem 2.36, we can just as well write

$$\begin{aligned} S &\leftrightarrows a.(T + S) + \mathbf{0} \\ T &\leftrightarrows \mathbf{1} + V \\ V &\leftrightarrows a.V + T. \end{aligned}$$

Now the transformation yields

$$\begin{aligned} S &\leftrightarrows a.A + \mathbf{0} \\ A &\leftrightarrows T + S \leftrightarrows \mathbf{1} + V + a.A + \mathbf{0} \leftrightarrows \mathbf{1} + a.V + T + a.A + \mathbf{0} \leftrightarrows \\ &\leftrightarrows \mathbf{1} + a.V + \mathbf{0} + a.A + \mathbf{0} \\ T &\leftrightarrows \mathbf{1} + V \leftrightarrows \mathbf{1} + a.V + T \leftrightarrows \mathbf{1} + a.V + \mathbf{0} \\ V &\leftrightarrows a.V + T \leftrightarrows a.V + \mathbf{1} + V \leftrightarrows a.V + \mathbf{1} + \mathbf{0}. \end{aligned}$$

The resulting specification is linear. We can simplify even further, and obtain

$$\begin{aligned} S &\leftrightarrows a.A \\ A &\leftrightarrows \mathbf{1} + a.V + a.A \\ T &\leftrightarrows \mathbf{1} + a.V \\ V &\leftrightarrows a.V + \mathbf{1}. \end{aligned}$$

This result can again be interpreted as a recursive specification, obtaining

$$\begin{aligned} S &= a.A \\ A &= \mathbf{1} + a.V + a.A \\ T &= \mathbf{1} + a.V \\ V &= a.V + \mathbf{1}. \end{aligned}$$

A first application of the notion of language equivalence is in determining the language of the automaton given by a recursive specification. We give an example using the distributive law of Table 3.

Consider the automaton given in Figure 2.27. The language contains a string $aaba$. We can write this string as a summand of the initial state S as follows:

$$S \approx aT + aW \approx aT + aaR \approx aT + aa(bW + \mathbf{1}) \approx aT + aabW + aa\mathbf{1} \approx$$

$$\approx aT + aabaR + aa\mathbf{1} \approx aT + aaba(bW + \mathbf{1}) + aa\mathbf{1} \approx aT + aababW + aaba\mathbf{1} + aa\mathbf{1}.$$

When we write down such a chain of equations, as soon as we use a law that is a language equivalence law, not a bisimulation law at one point in the chain, then we have to write $\approx$ everywhere.

This can be done in general: by following a path labeled w through the automaton, and expanding each time the states traversed, a summand $w\mathbf{1}$ can be split off. Conversely, if we can obtain a summand $w\mathbf{1}$ of the initial state in this way, this will come from a path labeled w from the initial state to a final state.

The derivation above can be simplified by using the symbol $\gtrsim$. For two terms x, y, $x \gtrsim y$ means that y is a summand of x. It can easily be defined as follows:

$$x \gtrsim y \qquad \Longleftrightarrow \qquad x \approx x + y.$$

In the exercises, we will show that this defines a partial order on terms. The derivation above now becomes:

$$S \approx aT + aW \gtrsim aW \approx aaR \approx aa(bW + \mathbf{1}) \approx aabW + aa\mathbf{1} \gtrsim aabW \approx$$

$$\approx aabaR \approx aaba(bW + \mathbf{1}) \approx aababW + aaba\mathbf{1} \gtrsim aaba\mathbf{1}.$$

We state the result in the form of a theorem.

Theorem 2.41. Let x be a term over MA with extra names $\mathcal{N}$. Then for all strings $w \in \mathcal{A}^*$:

$$w \in \mathcal{L}(x) \qquad \Longleftrightarrow \qquad x \gtrsim w\mathbf{1}.$$

We can use this theorem to show that any finite language is regular.

Theorem 2.42. Let $L \subseteq \mathcal{A}^*$ be finite. Then L is regular.

Proof. Enumerate $L = \{w_1, \ldots, w_n\}$. MA term $w_1\mathbf{1} + \ldots + w_n\mathbf{1}$ will accept this language. This means the language is regular. □

We can also define the operations of MA directly on automata, without going through the operational semantics.

1. $\mathbf{0}$ and $\mathbf{1}$ are shown in Figure 2.25.

2. Given an automaton with start state S, the automaton of $a.S$ is formed by adding a new start state $a.S$ and a new step $a.S \xrightarrow{a} S$.

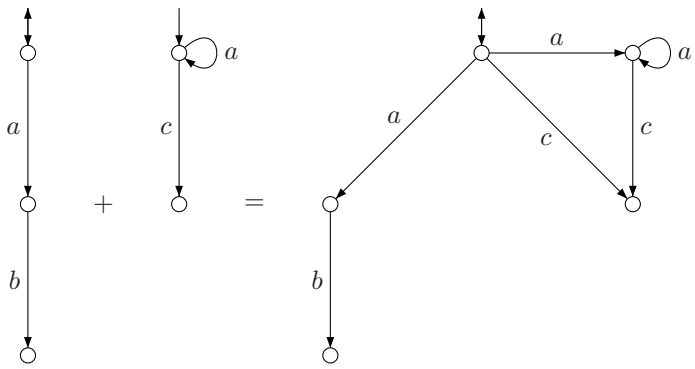

Figure 2.29: Alternative composition of two automata.

3. Suppose two automata are given with start states S, T, respectively. Add a new start state $S + T$, and add, for each step $S \xrightarrow{a} U$, a new step $S + T \xrightarrow{a} U$, and for each step $T \xrightarrow{a} V$, a new step $S + T \xrightarrow{a} V$. Also, $S + T \downarrow$ if one or both of $S \downarrow$ or $T \downarrow$. The original states S and T may become unreachable in the resulting automaton. See the example in Figure 2.29.

We finish the section with a simple theorem.

Theorem 2.43. Let L, L' be regular languages. Then also $L \cup L'$ is regular.

Proof. As L is regular, there is a recursive specification over MA with initial variable S such that $\mathcal{L}(S) = L$. As L' is regular, there is a recursive specification over MA with initial variable S' such that $\mathcal{L}(S') = L'$. Then $\mathcal{L}(S + S') = \mathcal{L}(S) \cup \mathcal{L}(S') = L \cup L'$, so also $L \cup L'$ is regular. □

Exercises

2.4.1 Find automata for the following terms by means of the operational rules:

(a) $a.(b.c.\mathbf{1} + b.\mathbf{0})$;

(b) $a.b.c.\mathbf{1} + a.b.\mathbf{0}$;

(c) $a.(b.\mathbf{1} + b.\mathbf{1}) + a.b.\mathbf{1}$;

(d) $a.(b.\mathbf{1} + \mathbf{0}) + a.b.\mathbf{1}$.

2.4.2 Given is the following recursive specification. Determine $\mathcal{L}(S)$.

$$\begin{aligned} S &= \mathbf{1} + a.A \\ A &= b.S. \end{aligned}$$

2.4.3 Find a recursive specification for the following languages, $\mathcal{A} = \{a\}$:

(a) $L = \{w \mid |w| \bmod 3 = 0\}$;

(b) $L = \{w \mid |w| \bmod 3 > 0\}$.

2.4.4 For the following recursive specifications, contruct an automaton by means of the operational rules. Next, turn them into linear form, and again construct an automaton by means of the operational rules.

$$\begin{aligned} S &= a.b.A \\ A &= b.a.B \\ B &= b.b.\mathbf{1} + a.A, \end{aligned}$$

$$S = a.b.S + b.a.S + \mathbf{1},$$

$$\begin{aligned} S &= a.S + b.T + \mathbf{0} \\ T &= S + a.(T + S). \end{aligned}$$

2.4.5 Construct a recursive specification for the following language:

$$L = \{a^n b^m \mid n \geq 2, m \geq 3\}.$$

2.4.6 Construct a recursive specification for the following language:

$$L = \{a^n b^m \mid n + m \text{ is even } \}.$$

2.4.7 Show that the automata generated by the following recursive specifications are bisimilar.

$$\begin{aligned} S &= a.T + a.U \\ T &= a.T + \mathbf{1} \\ U &= a.T + \mathbf{1} \end{aligned}$$

and

$$\begin{aligned} S &= a.T \\ T &= a.T + \mathbf{1}. \end{aligned}$$

2.4.8 Show that the automata generated by the following recursive specifications are bisimilar.

$$\begin{aligned} S &= a.T + b.U \\ T &= \mathbf{1} \\ U &= \mathbf{1} \end{aligned}$$

and

$$\begin{aligned} S &= a.T + b.T \\ T &= \mathbf{1}. \end{aligned}$$

2.4.9 Show that the automata generated by the following recursive specifications are bisimilar.

$$\begin{aligned} S &= a.T \\ T &= b.U + \mathbf{1} \\ U &= bU + \mathbf{1} \end{aligned}$$

and

$$\begin{aligned} S &= a.T \\ T &= b.T + \mathbf{1}. \end{aligned}$$

2.4.10 Which of the automata generated by the following recursive specifications are bisimilar?

$$\begin{aligned} S &= a.T + a.U \\ T &= \mathbf{1} \\ U &= \mathbf{0}, \end{aligned}$$

$$\begin{aligned} S &= a.T \\ T &= \mathbf{1}, \end{aligned}$$

$$\begin{aligned} S &= a.T + a.U \\ T &= \mathbf{1} \\ U &= \mathbf{1}, \end{aligned}$$

$$\begin{aligned} S &= a.T + a.U \\ T &= \mathbf{0} \\ U &= \mathbf{0}, \end{aligned}$$

and

$$\begin{aligned} S &= a.T \\ T &= \mathbf{0}. \end{aligned}$$

2.4.11 Which of the automata generated by the following recursive specifications are bisimilar?

$$\begin{aligned} S &= a.T \\ T &= a.T, \end{aligned}$$

$$\begin{aligned} S &= a.S + a.T \\ T &= \mathbf{1}, \end{aligned}$$

and

$$S = a.S.$$

2.4.12 (a) Prove that $x \gtrsim y$ iff there is a term z with $x \approx y + z$;
(b) Prove that $\gtrsim$ is reflexive: $x \gtrsim x$;
(c) Prove that $\gtrsim$ is anti-symmetric: $x \gtrsim y$ and $y \gtrsim x$ iff $x \approx y$;
(d) Prove that $\gtrsim$ is transitive: if $x \gtrsim y$ and $y \gtrsim z$, then $x \gtrsim z$;
(e) Prove that if $x \gtrsim y$, then $a.x \gtrsim a.y$;
(f) Prove that if $x \gtrsim y$ and $x' \gtrsim y'$, then $x + x' \gtrsim y + y'$.

2.4.13 Suppose language $L \cup L'$ is regular and L is regular. Show that L' need not be regular.

2.5 Deterministic automata

In general, there are many different automata that accept the same language. Sometimes, it can be advantageous to be able to find a language equivalent automaton of a particular form. In this section, we consider automata that are *total* and *deterministic*: this means that for every state and every element of the alphabet there is exactly one outgoing edge with this label.

Definition 2.44 (Total, deterministic automaton)**.** An automaton

$$M = (\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \downarrow)$$

is called *total* if for all $s \in \mathcal{S}$ and for all $a \in \mathcal{A}$ there is at least one $t \in \mathcal{S}$ with $s \xrightarrow{a} t$, and *deterministic* if for all $s \in \mathcal{S}$ and for all $a \in \mathcal{A}$ there is at most one $t \in \mathcal{S}$ with $s \xrightarrow{a} t$.

In a total automaton, each state has an outgoing edge for each label, in a deterministic automaton, each state cannot have two outgoing edges with the same label.

Example 2.45. Considering the automata displayed so far, the automata in Fig. 2.2, 2.28 and the third automaton in Fig. 2.23 are not deterministic, and only total if $\mathcal{A} = \{a\}$, the automata in Fig. 2.3, 2.4, 2.5, 2.7, 2.8, 2.19, 2.22 and the right-hand sides of Fig. 2.10, 2.15, 2.16, 2.18, 2.20, 2.21 are deterministic but not total, the automata in Fig. 2.26, 2.27 and the left-hand sides of Fig. 2.10, 2.15, 2.16, 2.18, 2.20, 2.21 are not deterministic and not total. In Fig. 2.23, the first, second and fourth automaton are deterministic, but only total if $\mathcal{A} = \{a\}$. In Fig. 2.25, all automata are deterministic, but the first two are only total if $\mathcal{A} = \emptyset$, and the third is not total.

Figure 2.30 shows an automaton that is total and deterministic. Notice that it accepts the same language as the automaton in Figure 2.5. The right-most state is called a *trap state*: any action from the alphabet can be executed at any time, but the state can never be exited, and it is not a final state.

By adding a trap state, any automaton can be made total, we can find a total automaton that accepts the same language. We will show that also, any automaton is language equivalent to a deterministic automaton. Notice that we cannot do this with bisimulation equivalence, it is easy to come up with an automaton that is not bisimulation equivalent to any total or deterministic automaton. See Fig. 2.31.

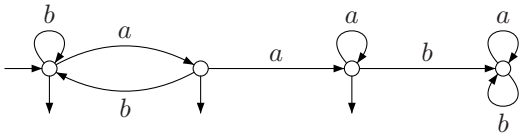

Figure 2.30: Total and deterministic automaton.

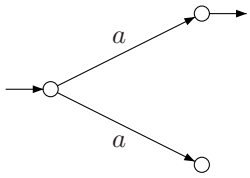

Figure 2.31: Not bisimilar to any total or deterministic automaton.

In this automaton, there is a deadlock state. Any bisimilar automaton will also contain a deadlock state, so cannot be total. The initial state has an a-step to two non-bisimilar states, so this must also be the case in any bisimilar automaton, so cannot be deterministic.

For every regular language, there exists a deterministic automaton that accepts it. This implies that (as far as the languages are concerned) general automata and deterministic automata are equally powerful: whenever a language is accepted by an automaton, it is also accepted by a deterministic automaton.

Theorem 2.46. Let L be a regular language. Then there is a deterministic automaton accepting L.

Proof. Let L be a regular language. As it is regular, there is an automaton $M = (\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \downarrow)$ accepting L. M is not necessarily deterministic.

If M has non-determinism, it has a state with more than one outgoing edge with the same label. Take a linear recursive specification representing M, so take a set of names $\mathcal{N}$ (with $S = \uparrow$) and an equation for each $P \in \mathcal{N}$. Then, the right-hand side of the equation of some P may have summands $a.Q$ and $a.R$ for different states Q, R. In such a case, apply the distributive law of Theorem 2.29, 'take the a outside the brackets', i.e. replace the two summands by one new summand $a.(Q + R)$. $Q + R$ will become a state of the transformed automaton. We apply the distributive law as often as possible, until no further application is possible, and moreover apply the commutative, associative and idempotence laws to remove brackets and superfluous summands. With this procedure, we obtain an automaton M'. Let us make the result explicit.

The automaton M' can be directly defined as follows:

1. The set of states of M' consists of sums of states of M. A sum over $\mathcal{N}$ is a state of M', if there is some string $w \in \mathcal{A}^*$ and a $P \in \mathcal{N}$ with $S \stackrel{w}{\longrightarrow\!\!\!\!\!\rightarrow} P$. Then, the sum consists of all these P, i.e. all $P \in \mathcal{N}$ satisfying $S \stackrel{w}{\longrightarrow\!\!\!\!\!\rightarrow} P$;

2. The alphabet is $\mathcal{A}$;

3. The initial state is S;

4. There is a transition labeled a from a sum state of M' to another sum state of M' exactly when there is such a transition from an element of the first sum to an element of the second sum in M;

5. A sum state of M' is a final state just in case one element is a final state in M.

By Theorem 2.29, the resulting automaton M' accepts the same language as M. □

Example 2.47. Let us give a simple example of the construction in the proof of this theorem. Given is the automaton on the left-hand side of Figure 2.32, with state names $\mathcal{S} = \{S, U, V, W\}$. As a linear specification, we get

$$\begin{aligned} S &= a.U + a.V \\ U &= b.S \\ V &= b.W + \mathbf{1} \\ W &= \mathbf{0}. \end{aligned}$$

We calculate $S \approx a.U + a.V \approx a.(U + V)$. From S, states U and V can be reached by an a-transition, so $U + V$ will be a state of the deterministic automaton, and $S \xrightarrow{a} U + V$. Then, consider state $U + V$. We have $U + V \approx b.S + b.W + \mathbf{1} \approx b.(S + W) + \mathbf{1}$, so we find state $S + W$ and $U + V \xrightarrow{b} S + W$. Continuing like this, we find the following set of equations.

$$\begin{aligned} S &\approx a.U + a.V \approx a.(U + V) \\ U + V &\approx b.S + b.W + \mathbf{1} \approx b.(S + W) + \mathbf{1} \\ S + W &\approx a.U + a.V + \mathbf{0} \approx a.(U + V) \end{aligned}$$

Considered as a recursive specification, this is a deterministic automaton. If, moreover, we want to get a total automaton, we add a trap state X, and provide missing edges:

$$\begin{aligned} S &\approx a.(U + V) + b.X \\ U + V &\approx b.(S + W) + \mathbf{1} + a.X \\ S + W &\approx a.(U + V) + b.X \\ X &\approx a.X + b.X \end{aligned}$$

The resulting specification can be given as follows:

$$\begin{aligned} S &= a.R + b.X \\ R &= b.P + \mathbf{1} + a.X \\ P &= a.R + b.X \\ X &= a.X + b.X \end{aligned}$$

The resulting automaton is displayed on the right-hand side of Figure 2.32.

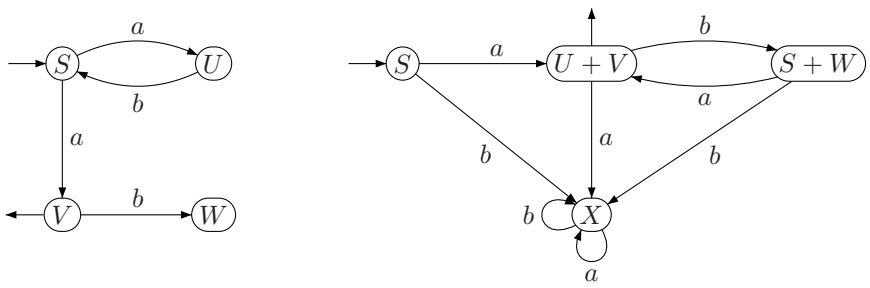

Figure 2.32: Finding a deterministic, and total automaton.

It should be remarked that the size of the resulting deterministic automaton can be much larger than the original automaton.

We also remark that the size of an automaton can be reduced as much as possible by identifying all states that have the same behavior. Thus, by identifying all states that can be colored the same, we get the minimal automaton that is bisimilar to a given automaton. Then, leaving out all states and edges that do not lead to a final state, and making the automaton deterministic, turns the automaton into the minimal automaton that is language equivalent to the given automaton.

We show the use of a deterministic and total automaton in the following theorem.

Definition 2.48. Let $L \subseteq \mathcal{A}^*$ be a language. The *complement* of L, denoted $\overline{L}$, is $\{w \in \mathcal{A}^* \mid w \notin L\}$.

Theorem 2.49. Let L, L' be regular languages.

1. The language $\overline{L}$ is regular.

2. The language $L \cap L'$ is regular.

Proof. For the first item, as L is regular, there is a deterministic and total automaton $M = (\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \downarrow)$ with $\mathcal{L}(M) = L$. Now consider the deterministic and total automaton

$$M' = (\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \mathcal{S} - \downarrow)$$

where the set of final states is the complement of the set of final states of the original automaton. Since both M and M' are deterministic and total, for every string w there is a unique path through them starting in the initial state. Now w is accepted by M' exactly when it is not accepted by M, or $\mathcal{L}(M') = \overline{\mathcal{L}(M)}$.

For the second item, consider the following formula:

$$L \cap L' = \overline{\overline{L} \cup \overline{L'}}.$$

Since the set of regular languages is closed under complementation and union, the right-hand side is regular. But then the left-hand side is also regular. □

Exercises

2.5.1 Find a deterministic and total automaton that is language equivalent to the automaton in Figure 2.2.

2.5.2 Find a deterministic and total automaton that is language equivalent to the automata in Figures 2.3 and 2.4.

2.5.3 Find a deterministic and total automaton that is language equivalent to each of the automata in Exercise 1 of Section 2.1.

2.5.4 Find a deterministic and total automaton that is language equivalent to each of the automata in Exercise 3 of Section 2.1.

2.5.5 Find a deterministic and total automaton that is language equivalent to each of the automata in Exercise 4 of Section 2.1.

2.5.6 Find a deterministic and total automaton that is language equivalent to each of the automata in Exercise 6 of Section 2.1.

2.5.7 Give a deterministic and total automaton for the following languages:

(a) $L = \{w \in \{a,b\}^* \mid$ the second from the last character is $a\ \}$

(b) $L = \{w \in \{a,b\}^* \mid \exists x, y \in \{a,b\}^*\ w = xabbaay \vee w = xbabay\}$

2.5.8 Suppose two deterministic and total automata are given with the same alphabet but disjoint state spaces. Define the cartesian product of the two automata, taking pairs of states as states in the product. A transition exists in the product if and only if it exists for the first components of the pair and also for the second components of the pair, and a pair is a final state if and only if both components are final. Formalize this definition, and show the language of the product is the intersection of the languages of the two automata.

2.5.9 Show the set of regular languages is closed under finite union and finite intersection, i.e. show that the union and the intersection of any number of regular languages is again regular.

2.5.10 Show the symmetric difference of two regular languages is a regular language. The symmetric difference of L and L' is defined by:

$$\{w \mid w \in L \text{ or } w \in L',\ \text{but not } w \in L \cap L'\}.$$

2.5.11 Suppose $L \cup L'$ is regular and L is finite. Show L' is regular.

2.5.12 For the statement below, decide whether it is true or false. If it is true, prove it. If not, give a counterexample. The alphabet is $\{a,b\}$.

If L is not regular, then $\overline{L}$ is not regular.

2.6 Automata with silent steps

So far, we considered automata where in every move, one input symbol is consumed. Further on, we will have need to describe the inner workings of a machine, and it will turn out to be useful to also allow moves where no input symbol is consumed. Such a step is called an *internal step* or a *silent step*, and we will always reserve the letter τ for such an internal move. An internal step is considered to be not observable. We might, however, be able to infer an internal move has taken place by other means.

Definition 2.50 (Automaton with internal moves)**.** An *automaton with internal moves* M is a quintuple $(\mathcal{S}, \mathcal{A} \cup \{\tau\}, \rightarrow, \uparrow, \downarrow)$ where:

1. $\mathcal{S}, \uparrow, \downarrow$ are as before;
2. $\mathcal{A}$ is a finite alphabet, and τ is a special symbol not in $\mathcal{A}$;
3. $\rightarrow \subseteq \mathcal{S} \times (\mathcal{A} \cup \{\tau\}) \times \mathcal{S}$ is the set of *transitions*.

A step $s \xrightarrow{\tau} t$ is called a *silent step*. Now the generalised transition relation $\twoheadrightarrow$ is inductively defined by adding the following clause to the two clauses in Definition 2.3:

3. For all $s, t, u \in \mathcal{S}$, if $s \xrightarrow{\tau} t$ and $t \overset{w}{\twoheadrightarrow} u$, then $s \overset{w}{\twoheadrightarrow} u$.

Thus, we see that the silent steps do not count when the label of a path is considered, it follows for instance that if $s \xrightarrow{\tau} t$, then $s \overset{\varepsilon}{\twoheadrightarrow} t$ (recall ε is the empty string). Based on this, the definition of acceptance and the language accepted by the automaton remains the same.

We also extend the Minimal Algebra with silent steps.

Definition 2.51. The Minimal Algebra is extended with silent steps by adding a prefix operator $\tau\_$, $\tau \notin \mathcal{A}$, and adding the following clause to Definition 2.25:

For all terms x, $\tau.x \xrightarrow{\tau} x$, a state named $\tau.x$ has a transition labeled τ to state x. See Table 5.

As a result, also the automaton of a term containing τ prefix operators can be determined.

$$\frac{}{\tau.x \xrightarrow{\tau} x}$$

Table 5: Operational rule for silent step.

To start with, we consider language equivalence.

An automaton with internal moves is a generalization of the notion of an automaton: every automaton is an automaton with internal moves (with no τ-steps at all), but an automaton containing silent steps is not an automaton in the original sense. The silent steps can be skipped in determining the language of an automaton. By this, we mean the following.

Theorem 2.52. Let x be a term over MA with extra names $\mathcal{N}$ and a recursive specification over these names. Then $\tau.x$ and x are language equivalent.

Proof. Let w be a string accepted by $\tau.x$. This means there is a path $\tau.x \stackrel{w}{\longrightarrow\!\!\!\!\!\rightarrow} y$ with $y \downarrow$. As the initial state $\tau.x$ is not a final state, this path must pass through state x, must start with step $\tau.x \stackrel{\tau}{\longrightarrow} x$. This means $x \stackrel{w}{\longrightarrow\!\!\!\!\!\rightarrow} y$, and w is also accepted by x.

The reverse direction is even simpler. □

We show the law for the silent step in Table 6.

$\tau.x \approx x$

Table 6: Language equivalence law for silent step.

Theorem 2.53. Let language L be accepted by an automaton with internal moves M. Then L is regular.

Proof. Let L be accepted by the automaton $M = (\mathcal{S}, \mathcal{A} \cup \{\tau\}, \rightarrow, \uparrow, \downarrow)$ with internal moves. We show L is also accepted by a normal automaton. This implies that L is regular. In order to construct this automaton, we use the law $\tau.P \approx P$.

Take a set of names $\mathcal{N}$ representing the states of M, and take a linear recursive specification of M. Now whenever $P \stackrel{\tau}{\longrightarrow} Q$ for $P, Q \in \mathcal{N}$, replace summand $\tau.Q$ in the right-hand side of the equation of P by Q, and then turn the resulting specification into a linear one, following Theorem 2.39. The result is a recursive specification not containing any τ prefix, and we call the resulting automaton M'.

The automaton M' can also be defined directly. M' has the same set of states and the same initial state as M. Whenever $P \stackrel{\tau}{\longrightarrow} Q \stackrel{a}{\longrightarrow} R$ in M ($a \neq \tau$), add $P \stackrel{a}{\longrightarrow} R$ in M', and whenever $P \stackrel{\tau}{\longrightarrow} Q \downarrow$ in M, add $P \downarrow$ in M', and do this repeatedly until no further steps or final states are obtained. Then, erase all τ-steps and remove unreachable states, resulting in M'.

To see that M' is language equivalent to M, take a string $w \in \mathcal{L}(M)$. Thus, w is the label of a path through M from the initial state to a final state. We can also trace w through M'. For, whenever the path through M uses a transition $s \stackrel{\tau}{\longrightarrow} t$, follow the path further until a non-τ step $t' \stackrel{a}{\longrightarrow} t''$ is reached. Then, in M', we can directly take $s \stackrel{a}{\longrightarrow} t''$. If there is no non-$\tau$-step further down the path, then after a number of τ-steps a final state is reached, and in M' we have $s \downarrow$. □

Example 2.54. We give an example of the construction in the proof of this theorem. Consider the automaton at the left-hand side of Figure 2.33. It yields the recursive specification

$$\begin{aligned} S &= a.W + \tau.T \\ T &= a.U \\ U &= \tau.U + b.V + \tau.T \\ V &= \mathbf{0} \\ W &= \mathbf{1}. \end{aligned}$$

We calculate:

$$\begin{aligned} S &\approx a.W + \tau.T \approx a.W + T \approx a.W + a.U \\ U &\approx \tau.U + b.V + \tau.T \approx U + b.V + T \approx \mathbf{0} + b.V + a.U \\ V &\approx \mathbf{0} \\ W &\approx \mathbf{1} \end{aligned}$$

As T is not a reachable state in the resulting automaton, it can be left out. The resulting specification is as follows:

$$\begin{aligned} S &= a.W + a.U \\ U &= \mathbf{0} + b.V + a.U \\ V &= \mathbf{0} \\ W &= \mathbf{1} \end{aligned}$$

We show the resulting automaton on the right-hand side of Figure 2.33. Note that the τ-loop on U is just omitted.

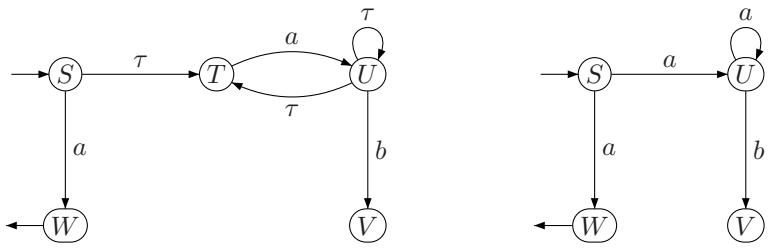

Figure 2.33: Finding an automaton without silent steps.

As an example of an automaton with internal moves, consider an automaton that accepts decimal numbers. A decimal number starts with an optional + or − sign, then has a string of digits followed by a decimal point again followed by a string of digits. One of these strings of digits can be empty, but not both. We display the automaton in Figure 2.34.

An automaton with internal moves is useful to prove the following result. Define $L^R = \{w^R \mid w \in L\}$, the *reverse* of L.

Theorem 2.55. Let language L be regular. Then L^R is regular.

Proof. As L is regular, we know there is an automaton $M = (\mathcal{S}, \mathcal{A}, \uparrow, \downarrow, \rightarrow)$ with $\mathcal{L}(M) = L$. Convert M into an automaton with a single final state, by adding a state $s \notin \mathcal{S}$, and adding edges $t \xrightarrow{\tau} s$ for each final state t of M. It is obvious that this automaton accepts the same language as M. Next, turn the initial state $\uparrow$ into a final state, turn the final state s into the initial state, and reverse the direction of all edges. The modified automaton accepts the reversal of L. □

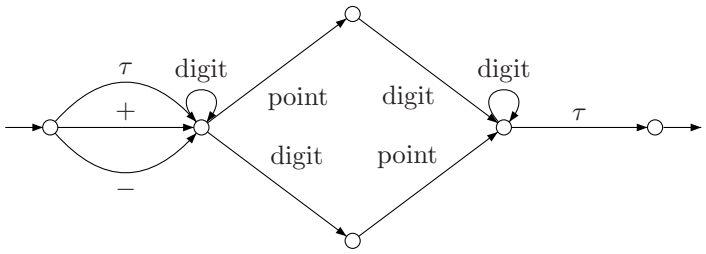

Figure 2.34: Decimal numbers.

When we are dealing with automata with internal moves, some of the notions considered earlier have to reinterpreted. An example of this is the notion of determinism. In defining when an automaton with internal moves is deterministic, the silent moves have to be disregarded, as stated next.

Definition 2.56 (Deterministic automaton)**.** An automaton with internal moves

$$M = (\mathcal{S}, \mathcal{A} \cup \{\tau\}, \rightarrow, \uparrow, \downarrow)$$

is called *deterministic* if for all $s \in \mathcal{S}$ and for all $a \in \mathcal{A}$ there is at most one $t \in \mathcal{S}$ with $s \stackrel{a}{\longrightarrow\!\!\!\!\!\rightarrow} t$.

Thus, all paths from a state s that have only the visible label a must lead to the same state. In a similar way, the notion of totality can be redefined.

Exercises

2.6.1 Draw the automata of the following terms by means of the operational rules. Then, determine the language of the resulting automata.

(a) $\tau.a.\mathbf{1} + \tau.b.\mathbf{0}$;
(b) $\tau.(\tau.a.a.\mathbf{1} + b.\tau.\mathbf{1})$;
(c) $\tau.\mathbf{1} + \tau.\mathbf{0}$.

2.6.2 Draw an automaton for the following recursive specification by means of the operational rules. Then, construct a language equivalent automaton without τ-steps.

$$\begin{aligned} S &= S + \tau.S + \tau.T + a.S \\ T &= b.T + \tau.S + \tau.\mathbf{1}. \end{aligned}$$

2.6.3 Draw an automaton for the following recursive specification by means of the operational rules. Then, construct a language equivalent automaton without τ-steps.

$$\begin{aligned} S &= a.S + b.T + c.U \\ T &= \tau.S + a.T + b.U \\ U &= \tau.T + a.U + c.S. \end{aligned}$$

2.6.4 Draw an automaton for the following recursive specification by means of the operational rules. Then, construct a language equivalent automaton without τ-steps.

$$\begin{aligned} S &= \tau.T + \tau.U + b.T + c.U \\ T &= a.S + b.U + c.S + c.T \\ U &= \mathbf{0}. \end{aligned}$$

2.6.5 Construct an automaton for the following languages. Use τ-moves to simplify your design.

(a) The set of strings consisting of zero or more a's followed by zero or more b's;

(b) The set of strings that consist of either ab repeated one or more times or aba repeated one or more times.

2.7 Branching bisimulation

In the previous section, we found that the treatment of silent steps in language equivalence is rather straightforward. Here, we will find that treatment in bisimulation needs carefulness. We may not be able to observe τ-steps directly, but in many cases, it can be observed indirectly that a τ-step has taken place. Consider the three pictures in Figure 2.35, each showing part of an automaton. The top nodes may have some incoming edges, but no outgoing edges that are not shown, the bottom nodes may have some other incoming or outgoing edges.

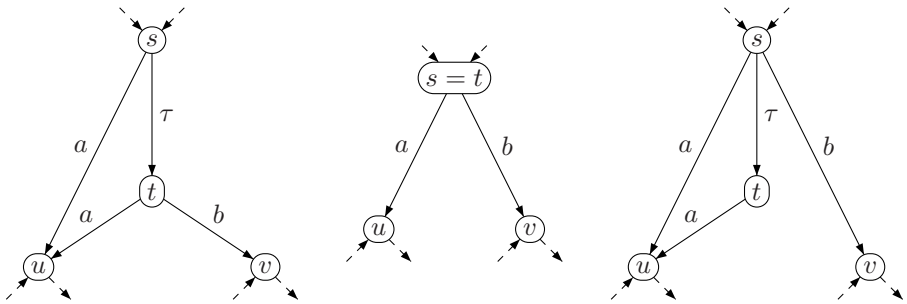

Figure 2.35: Parts of an automaton, a silent step that is inert and not inert.

The middle part is obtained from the part on the left, by identifying the nodes s and t and leaving out the τ-step. It is also obtained from the part on the right, by identifying the nodes s and t and leaving out the τ-step. We will argue the first procedure is valid for bisimulation, while the second is not.

The τ on the left leads from a node with just an outgoing a-step to u to a node with both an a to u and a b to v. By executing the τ, we gain the option of doing a b to v, but we do not lose the option of doing an a to u. We can

argue the b to v was already possible from s by going through the silent τ. This τ is called *inert*: an option may be gained, but no option is lost.

The τ on the right leads from a node with both an a to u and a b to v to a node with just an a to u. By executing the τ, we lose the option of doing a b to v, this is not possible any more from t. This τ is called *not inert*: an option is lost by executing it.

If in Figure 2.35 we remove all a-steps, then the part on the right contains a deadlock node t, but the parts in the middle and left do not contain a deadlock node. As bisimulation should preserve deadlocks, this again says that the part on the right cannot be identified with the part in the middle. Thus, the not inert τ on the right cannot be removed.

The notion of bisimulation we will formulate will allow removal of inert τ's by identification of the nodes it is between, but all τ's that are not inert cannot be removed.

We show two examples of inert τ-steps in Figure 2.36. In the picture on the left, the top node does not have any outgoing steps except for the τ-step.

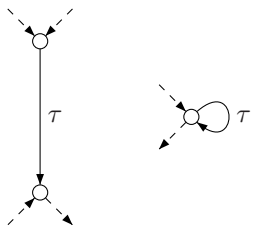

Figure 2.36: Inert τ-steps.

Definition 2.57 (Branching bisimilarity)**.** A binary relation R between the sets of states $\mathcal{S}, \mathcal{S}'$ of two automata M, M' is a *branching bisimulation* relation if and only if the following conditions hold:

1. R relates the reachable states;

2. R relates the initial states;

3. whenever sRs' and $s \xrightarrow{a} t$ for some $a \in \mathcal{A}$, then there are states $\widehat{s'}$ and t' in $\mathcal{S}'$ such that $s' \xrightarrow{\varepsilon} \widehat{s'}$ and $\widehat{s'} \xrightarrow{a} t'$ and both $sR\widehat{s'}$ and tRt'; note that the path from s' to $\widehat{s'}$ contains a number of τ-steps (0 or more); this is the transfer condition for visible actions from left to right, see Figure 2.37;

4. whenever sRs' and $s' \xrightarrow{a} t'$ for some $a \in \mathcal{A}$, then there are states $\widehat{s}$ and t in $\mathcal{S}$ such that $s \xrightarrow{\varepsilon} \widehat{s}$ and $\widehat{s} \xrightarrow{a} t$ and both $\widehat{s}Rs'$ and tRt'; this is the transfer condition for visible actions from right to left;

5. whenever sRs' and $s \xrightarrow{\tau} t$, then there are states $\widehat{s'}$ and t' in $\mathcal{S}'$ such that $s' \xrightarrow{\varepsilon} \widehat{s'}$ and either $\widehat{s'} = t'$ or $\widehat{s'} \xrightarrow{\tau} t'$ and both $sR\widehat{s'}$ and tRt'; this is the transfer condition for silent steps from left to right, see Figure 2.38;

6. whenever sRs' and $s' \xrightarrow{\tau} t'$, then there are states $\widehat{s}$ and t in $\mathcal{S}$ such that $s \xrightarrow{\varepsilon} \!\!\!\!\rightarrow \widehat{s}$ and either $\widehat{s} = t$ or $\widehat{s} \xrightarrow{\tau} t$ and both $\widehat{s}Rs'$ and tRt'; this is the transfer condition for silent steps from right to left;

7. whenever sRs' and $s \downarrow$, then there is a state $\widehat{s'} \in \mathcal{S}'$ such that $s' \xrightarrow{\varepsilon} \!\!\!\!\rightarrow \widehat{s'}$, $\widehat{s'} \downarrow$, and $sR\widehat{s'}$; this is the transfer condition for final states from left to right, see Figure 2.39;

8. whenever sRs' and $s' \downarrow$, then there is a state $\widehat{s} \in \mathcal{S}$ such that $s \xrightarrow{\varepsilon} \!\!\!\!\rightarrow \widehat{s}$, $\widehat{s} \downarrow$, and $\widehat{s}Rs'$; this is the transfer condition for final states from right to left.

Two automata M, M' are *branching bisimulation equivalent* or *branching bisimilar*, notation $M \leftrightarrows_{\mathrm{b}} M'$, if and only if there is a branching bisimulation relation R between M and M'.

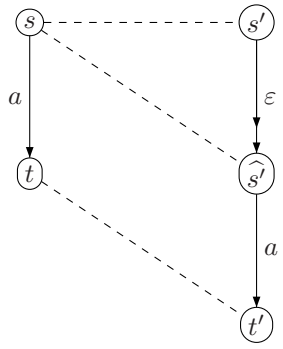

Figure 2.37: Transfer condition for visible actions from left to right.

The 'normal' notion of bisimulation we had prior to the formulation of branching bisimulation will from now be called *strong bisimulation*, in order not to be confused with branching bisimulation. We can remark that any two automata that are strongly bisimilar, are necessarily also branching bisimilar.

Theorem 2.58 (Equivalence)**.** Branching bisimilarity is an equivalence.

Proof. Let M, M', M'' be automata with silent steps. First, the relation $R = \{(s, s) \mid s \in \mathcal{S}, s \text{ reachable}\}$ is obviously a branching bisimulation relation. This proves that $M \leftrightarrows_{\mathrm{b}} M$. Second, if R is a branching bisimulation relation for $M \leftrightarrows_{\mathrm{b}} M'$, then the relation $R' = \{(s', s) \mid sRs'\}$ is a branching bisimulation relation as well, implying $M' \leftrightarrows_{\mathrm{b}} M$. Third, assume that $M \leftrightarrows_{\mathrm{b}} M'$ and $M' \leftrightarrows_{\mathrm{b}} M''$. Let R_1 and R_2 be branching bisimulation relations relating the reachable states of M, M' and M', M'', respectively. Then the relation composition $R_1 \circ R_2$ is a branching bisimulation relation that shows $M \leftrightarrows_{\mathrm{b}} M''$. □

We can also formulate branching bisimulation in terms of colors, as follows. Again, color all the nodes of an automaton, and we get colored paths that

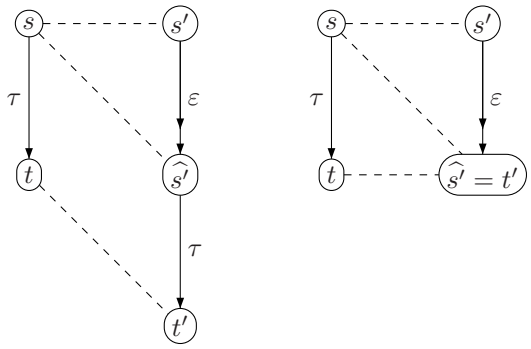

Figure 2.38: Transfer condition for silent steps from left to right.

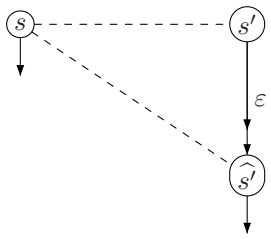

Figure 2.39: Transfer condition for final states from left to right.

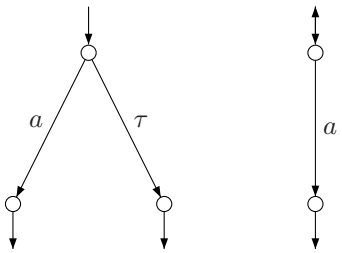

Figure 2.40: Branching bisimilar automata?

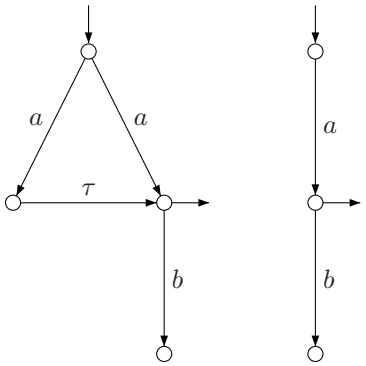

Figure 2.41: Branching bisimilar automata?

consist of colors and steps in alternation. Now, whenever such a path contains a part $c \xrightarrow{\tau} c$, so a τ-step between nodes of the same color, then this part can be replaced by c, the τ-step can be left out. Such reduced colored paths are called *abstract* colored paths. Call a coloring *abstract consistent* if all nodes of the same color have the same abstract colored paths. Now two nodes can be related by a branching bisimulation exactly when there is an abstract consistent coloring that gives them the same color. An inert τ can be characterized as a τ between two nodes of the same color.

Thus, branching bisimulation is an equivalence, but it is not a congruence. A simple example serves to make this point: we have $\tau.\mathbf{1} \leftrightarrows_{\mathrm{b}} \mathbf{1}$, but $\tau.\mathbf{1}+a.\mathbf{1}$ is not branching bisimilar to $\mathbf{1}+a.\mathbf{1}$ (executing the τ will disallow the a-action). This problem can be fixed, but complicates the matter further. This is why we will not present a solution for the problem, but refrain from doing any calculations in bisimulation on terms with τ's. Instead, we will only use branching bisimulation on automata or transition systems.

Exercises

2.7.1 Are the pairs of automata in Figures 2.40, 2.41, 2.42, 2.43, 2.44, 2.45, 2.46, 2.47 branching bisimilar? If so, give a branching bisimulation between the two automata; otherwise, explain why they are not branching bisimilar.

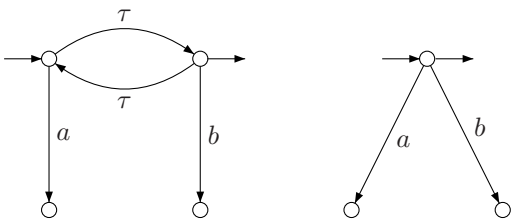

Figure 2.42: Branching bisimilar automata?

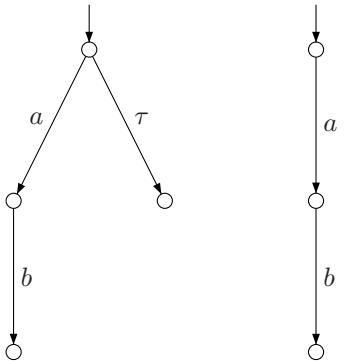

Figure 2.43: Branching bisimilar automata?

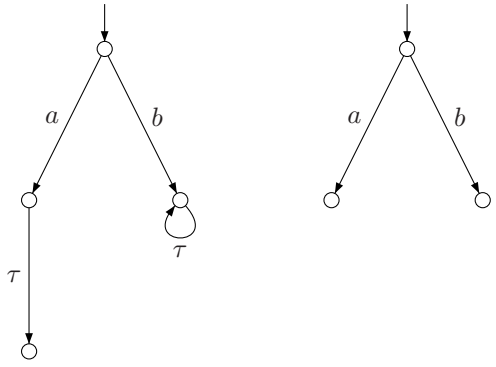

Figure 2.44: Branching bisimilar automata?

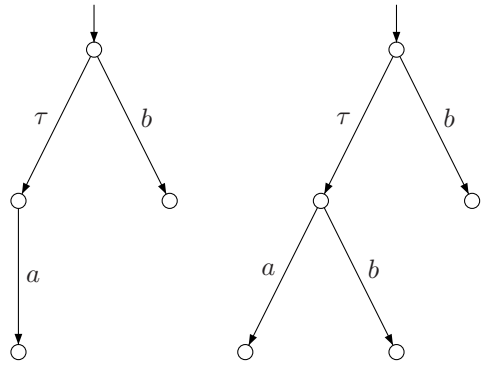

Figure 2.45: Branching bisimilar automata?

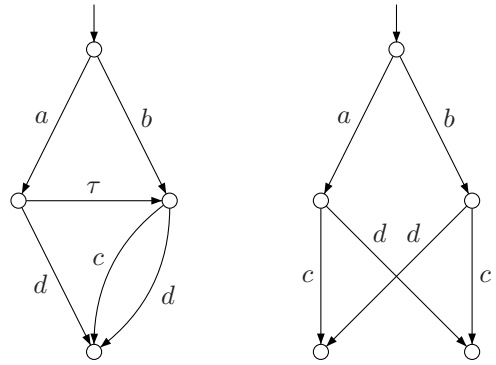

Figure 2.46: Branching bisimilar automata?

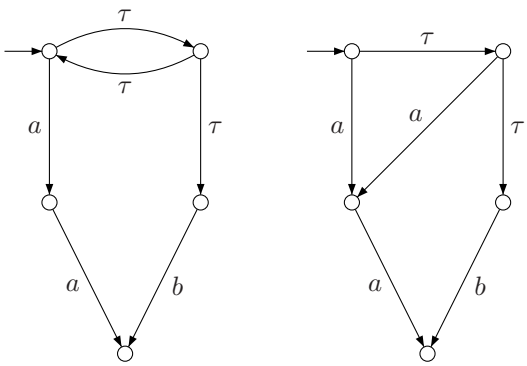

Figure 2.47: Branching bisimilar automata?

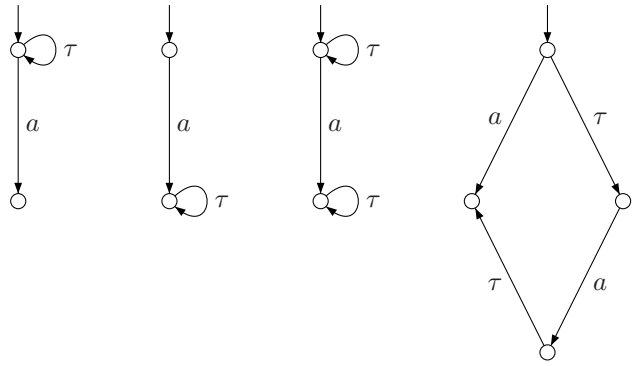

Figure 2.48: All automata are branching bisimilar.

2.7.2 Give a branching bisimulation between any two of the automata in Figure 2.48.

2.7.3 Which of the transition systems of Exercises 1 and 2 are deadlock free?

2.8 Identifying non-regular languages

In Theorem 2.8 and Exercise 24 of Section 2.1, we showed that a certain language was not regular. The idea of the proof in both cases was the so-called *pigeonhole principle*: if you put a number of things greater than n into n places, then at least one place will contain more than one thing.

In this section, we will meet a general technique to show that some language is not regular: the *pumping lemma*. The lemma only works for infinite languages, but that does not matter, as all finite languages are regular anyway (see Proposition 2.42).

Theorem 2.59 (Pumping lemma for regular languages)**.** Let L be an infinite regular language. Then there is a positive integer m such that any $w \in L$ with $|w| \geq m$ can be written as $w = xyz$ with $|xy| \leq m$ and $|y| \geq 1$ and for all $i \geq 0$ we have that $xy^iz \in L$.

Presented symbolically:

L infinite and regular $\Longrightarrow$

$$\exists m\,(m > 0 :\ \forall w\,(w \in L\ \wedge\ |w| \geq m : \exists x, y, z\,(w = xyz\ \wedge\ |xy| \leq m\ \wedge\ |y| \geq 1 :$$

$$\forall i\,(i \geq 0 : xy^iz \in L)))).$$

Proof. We use the pigeonhole principle. Since L is regular, there is an automaton M accepting L. The set of states of M is finite, say M has m states. Since L is infinite, we can find strings in L that are arbitrarily long. Take a string $w \in L$ with $|w| \geq m$. Thus, there is a path through M from the initial state to a final state with label w.

Now this path goes through at least $m+1$ states of M, so it must go through some state it has visited before. Take the first state s that is repeated in this

path. The path traced by w can now be broken up into three parts: first, the part from the initial state until s is visited for the first time, say this has label x, the part from s until s is visited for the second time, say this has label y, and finally the part from s to a final state, say this has label z. Then $w = xyz$ and $|xy| \leq m$ and $|y| \geq 1$.

Moreover, by taking the first part to s followed by the third part to a final state, we see $xz \in L$. By taking the first part, followed by the second part twice followed by the third part, we see $xy^2z \in L$, and so on. □

We see that if we have an infinite regular language L, then if we take a string in L that is sufficiently long, then we can break this string into three parts, and the middle part of the string can be pumped arbibrarily many times, staying inside L.

We often apply this theorem by using the contrapositive:

$$\forall m\,(m > 0 :\ \exists w\,(w \in L\ \wedge\ |w| \geq m : \forall x, y, z\,(w = xyz\ \wedge\ |xy| \leq m\ \wedge\ |y| \geq 1 :$$

$$\exists i\,(i \geq 0 : xy^iz \in L))))$$

$\Longrightarrow L$ is not infinite or not regular.

Example 2.60. $L = \{a^nb^n \mid n \geq 0\}$ is not regular. For, if L were regular, then the pumping lemma applies. Take the value m given by the pumping lemma. Take string $w = a^mb^m \in L$. Then write $w = xyz$ as given by the pumping lemma. As $|xy| \leq m$, x and y consist entirely of a's. Suppose $|y| = k > 0$. Take $i = 0$. Then $xz = a^{m-k}b^m \in L$, and this is a contradiction. So the assumption that L was regular was wrong. Thus, L is not regular.

Example 2.61. $L = \{ww^R \mid w \in \{a, b\}^*\}$ is not regular. For, if L were regular, then the pumping lemma applies. Take the value m given by the pumping lemma. Choose string $w = a^mb^mb^ma^m \in L$. Then write $w = xyz$ as given by the pumping lemma. As $|xy| \leq m$, x and y consist entirely of a's. Suppose $|y| = k > 0$. Take $i = 0$. Then $xz = a^{m-k}b^mb^ma^m \in L$, and this is a contradiction. So the assumption that L was regular was wrong. Thus, L is not regular.

Example 2.62. $L = \{w \in \{a, b\}^* \mid \#_a(w) < \#_b(w)\}$ is not regular. For, if L were regular, then the pumping lemma applies. Take the value m given by the pumping lemma. Choose string $w = a^mb^{m+1} \in L$. Then write $w = xyz$ as given by the pumping lemma. As $|xy| \leq m$, x and y consist entirely of a's. Suppose $|y| = k > 0$. This time, taking $i = 0$ will not work, and we need to consider $i = 2$. We see $xy^2z = a^{m+k}b^{m+1} \in L$, and this is a contradiction. So the assumption that L was regular was wrong. Thus, L is not regular.

Example 2.63. $L = \{(ab)^na^k \mid n > k, k \geq 0\}$ is not regular. For, if L were regular, then the pumping lemma applies. Take the value m given by the pumping lemma. Choose string $w = (ab)^{m+1}a^m \in L$. Then write $w = xyz$ as given by the pumping lemma. As $|xy| \leq m$, x and y are in the ab-repeating part. We now have to do a case analysis on y. As y is non-empty, it contains at least one symbol. If y is just a, we can take $i = 0$ and we find $xz \notin L$. Likewise if y is just b. If y is ab or ba, then, again taking $i = 0$, we see $xz = (ab)^mb^m$, which cannot be in L. In all other cases, y contains at least one a and one b,

and so xz will be a substring of $(ab)^m a^m$, where some part of the initial $(ab)^m$ part is left out.

In all cases $xz \in L$ leads to a contradiction. So the assumption that L was regular was wrong. Thus, L is not regular.

Exercises

2.8.1 Show that the language $\{w \mid \#_a(w) = \#_b(w)\}$ is not regular.

2.8.2 Prove that the following languages are not regular.

(a) $\{a^n b^l a^k \mid k \geq n + l\}$;
(b) $\{a^n b^l a^k \mid k \neq n + l\}$;
(c) $\{a^n b^l a^k \mid n = l \text{ or } l \neq k\}$;
(d) $\{a^n b^k \mid n \leq k\}$;
(e) $\{w \mid \#_a(w) \neq \#_b(w)\}$;
(f) $\{ww \mid w \in \{a, b\}^*\}$.

2.8.3 Determine whether or not the following languages are regular.

(a) $\{a^n \mid n \text{ is a prime number}\}$;
(b) $\{a^n \mid n \text{ is not a prime number}\}$;
(c) $\{a^n \mid n = k^2 \text{ for some } k \geq 0\}$;
(d) $\{a^n \mid n = 2^k \text{ for some } k \geq 0\}$.

2.8.4 Determine whether or not the following languages are regular.

(a) $\{a^n b^k \mid n \geq 100, k \leq 100\}$;
(b) $\{a^n b^k \mid n \leq k \leq 2n\}$;
(c) $\{a^n b^l a^k \mid n + l + k > 5\}$;
(d) $\{a^n b^l a^k \mid n > 5, l > 3, k \leq l\}$;
(e) $\{ww^R \mid w \in \{a, b\}^*\}$.

Chapter 3

Extending the Algebra

In this chapter, we extend the Minimal Algebra of the previous chapter with extra operators. These operators will give us extra notations for finite automata and extra ways to compose finite automata. We will be able to describe interacting automata. We will be able to specify nonregular processes.

3.1 Sequential Composition

We start by considering sequential composition.

Definition 3.1. The Sequential Algebra SA extends MA with the binary operator $\_ \cdot \_$ denoting *sequential composition*.

The operational rules for sequential composition are given in Table 7.

$$\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \qquad \frac{x\downarrow \quad y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'} \qquad \frac{x\downarrow \quad y\downarrow}{x \cdot y \downarrow}$$

Table 7: Additional operational rules for SA ($a \in \mathcal{A}$).

These rules can be explained as follows: term $x \cdot y$ can start by executing a step from x (resulting in the remainder of x followed by y). The other possibility only occurs when x is a final state. Then, we can start by executing a step from y (thereby exiting x) or it is a final state when also y is final.

We see that the reachable states of a term $x \cdot y$ are all states $x' \cdot y$, where x' is reachable from x plus, in case x has a final state, all states y' reachable from y (except maybe y itself). Thus, the set of reachable states is finite. This implies the following theorem.

Theorem 3.2. Let x, y be expressions over Minimal Algebra, possibly containing added names. Then $x \cdot y$ is a finite automaton.

The theorem implies that the sequential composition of two regular processes is again a regular process.

Notice that this theorem does *not* imply that every expression over SA with added names denotes a finite automaton. To see this, consider the following example.

Example 3.3. Consider the recursive equation $S = \mathbf{1} + a.S \cdot b.\mathbf{1}$. From the second summand we obtain $S \xrightarrow{a} S \cdot b.\mathbf{1}$. Using $S \downarrow$ we obtain $S \cdot b.\mathbf{1} \xrightarrow{b} \mathbf{1}$. Repeating this gives $S \cdot b.\mathbf{1} \xrightarrow{a} S \cdot b.\mathbf{1} \cdot b.\mathbf{1}$ and $S \cdot b.\mathbf{1} \cdot b.\mathbf{1} \xrightarrow{b} \mathbf{1} \cdot b.\mathbf{1}$. Repeating this a number of times yields the transition system of the simple counter 2.17. Thus, this recursive equation does not yield a regular process.

So, allowing sequential composition inside the recursion can yield a nonregular process. On the other hand, if we take two recursive specifications over MA, then their sequential composition will be regular.

We can determine the language that is accepted by a term over SA.

Theorem 3.4. Let x, y be SA-terms.

$$\mathcal{L}(x \cdot y) = \{uv \in \mathcal{A}^* \mid u \in \mathcal{L}(x) \;\&\; v \in \mathcal{L}(y)\}$$

Proof. Straightforward. □

Also, we can directly determine the automaton that results from the sequential composition of two automata. Take two names $S, T \in \mathcal{N}$ and suppose S, T each have a *linear* recursive specification with a disjoint set of names. Then the automaton $S \cdot T$ has the set of states $U \cdot T$, for each name U of the specification of S, plus the set of names of T (omitting T itself if this becomes not reachable). Next, $U \cdot T \xrightarrow{a} U' \cdot T$ just in case $U \xrightarrow{a} U'$, $U \cdot T \xrightarrow{a} V$ (for V a name of T) just in case $U \downarrow$ and $T \xrightarrow{a} V$ and $U \cdot T \downarrow$ just in case $U \downarrow$ and $T \downarrow$. We keep the transitions and terminations of the second automaton.

Consider the example in Figure 3.1.

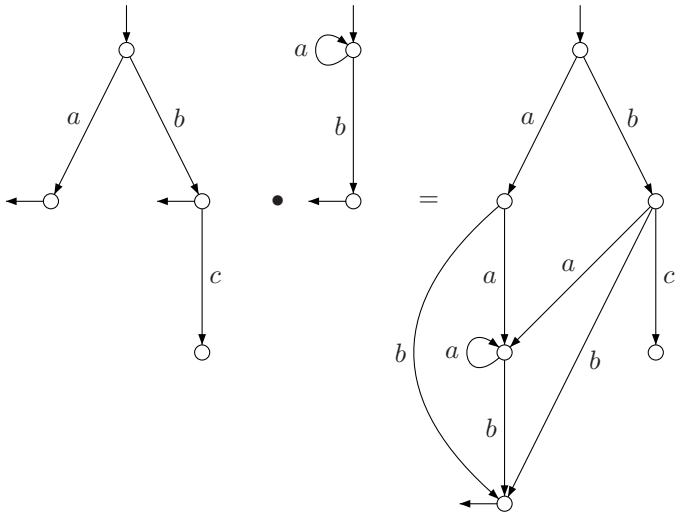

Figure 3.1: Sequential composition of two automata.

We can also define sequential composition of languages in the obvious way:

$$L \cdot L' = \{uv \mid u \in L, v \in L'\}.$$

This is called *concatenation* of languages. We see immediately that if L, L' are regular, then also $L \cdot L'$ is regular.

Now we investigate laws for sequential composition.

Theorem 3.5. The following laws hold for all terms over SA, possibly containing added names.

1. $(a.x) \cdot y \leftrightarrows a.(x \cdot y)$,
2. $(x \cdot y) \cdot z \leftrightarrows x \cdot (y \cdot z)$,
3. $(x + y) \cdot z \leftrightarrows x \cdot z + y \cdot z$,
4. $\mathbf{1} \cdot x \leftrightarrows x$,
5. $x \cdot \mathbf{1} \leftrightarrows x$,
6. $\mathbf{0} \cdot x \leftrightarrows \mathbf{0}$.

Proof. 1. $(a.x) \cdot y \leftrightarrows a.(x \cdot y)$, action prefix distributes over sequential composition. The automaton of $(a.x) \cdot y$ is exactly the same as the automaton of $a.(x \cdot y)$, apart from the name of the initial state, and so the automata are bisimilar. The initial state is not a final state, and only has step $(a.x) \cdot y \xrightarrow{a} x \cdot y$ respectively $a.(x \cdot y) \xrightarrow{a} x \cdot y$.

2. $(x \cdot y) \cdot z \leftrightarrows x \cdot (y \cdot z)$, sequential composition is associative. The reachable states of $(x \cdot y) \cdot z$ are $(x' \cdot y) \cdot z$, for each x' reachable from x, plus, in case x has a final state, all states reachable from $y \cdot z$ except maybe $y \cdot z$ itself. Now, for each state of the form $(x' \cdot y) \cdot z$, there is a corresponding state $x' \cdot (y \cdot z)$ in $x \cdot (y \cdot z)$ The other states match, and transitions and final states coincide, and so the automata are bisimilar.

3. $(x + y) \cdot z \leftrightarrows x \cdot z + y \cdot z$, sequential composition distributes from the right over alternative composition. The automaton of $(x+y) \cdot z$ is exactly the same as the automaton of $x \cdot z + y \cdot z$, apart from the name of the initial state, and so the automata are bisimilar. For, if $(x + y) \cdot z \xrightarrow{a} p$ for certain $a \in \mathcal{A}$ and term p, then the first possibility is $x + y \xrightarrow{a} q$ and p is of the form $q \cdot z$. But then either $x \xrightarrow{a} q$ or $y \xrightarrow{a} q$ and it follows that $x \cdot z + y \cdot z \xrightarrow{a} q \cdot z$. The second possibility is that $x + y$ is a final state and $z \xrightarrow{a} p$ and it follows that either x or y is a final state and also $x \cdot z + y \cdot z \xrightarrow{a} p$. Next, if $(x+y) \cdot z \downarrow$, then we must have $x+y \downarrow$ and $z \downarrow$. It follows that $x \downarrow$ or $y \downarrow$, so $x \cdot z + y \cdot z \downarrow$. The reverse direction is similar.

4. $\mathbf{1} \cdot x \leftrightarrows x$, $\mathbf{1}$ is a left unit element for sequential composition. The automaton of $\mathbf{1} \cdot x$ is exactly the same as the automaton of x, apart from the name of the initial state,and so the automata are bisimilar. For, $\mathbf{1} \cdot x \xrightarrow{a} y$ just in case $x \xrightarrow{a} y$, and $\mathbf{1} \cdot x \downarrow$ just in case $x \downarrow$.

5. $x \cdot \mathbf{1} \leftrightarrows x$, $\mathbf{1}$ is a right unit element for sequential composition. The states of $x \cdot \mathbf{1}$ are $x' \cdot \mathbf{1}$ for all x' reachable from x. $x' \cdot \mathbf{1}$ can take a step exactly when x' can and is a final state exactly when x' is. Thus, the automata are bisimilar.

6. $\mathbf{0} \cdot x \leftrightarrows \mathbf{0}$, $\mathbf{0}$ is not a final state and cannot execute any step, so x in $\mathbf{0} \cdot x$ can never be reached, and $\mathbf{0} \cdot x$ has just one state, the initial state, not final. The automata are bisimilar.

□

Now we consider laws that do change the automaton, but preserve language equivalence.

Theorem 3.6. Let x, y, z be three terms over SA. Then automata $x \cdot \mathbf{0}$ and $\mathbf{0}$ accept the same language. Also $x \cdot (y + z)$ and $x \cdot y + x \cdot z$ accept the same language.

Proof. For the first part, note that all states of the automaton of $x \cdot \mathbf{0}$ have the form $x' \cdot \mathbf{0}$, where x' is reachable from x. None of these states is a final state, $\downarrow = \emptyset$. It follows that $\mathcal{L}(x \cdot \mathbf{0}) = \emptyset = \mathcal{L}(\mathbf{0})$.

For the second part, consider a string $w \in \mathcal{L}(x \cdot (y + z))$. We can write $w = uv$, where $u \in \mathcal{L}(x)$ and $v \in \mathcal{L}(y + z)$ (u or v may be the empty string). Since $\mathcal{L}(y+z) = \mathcal{L}(y) \cup \mathcal{L}(z)$, $v \in \mathcal{L}(y)$ or $v \in \mathcal{L}(z)$. It follows that $uv \in \mathcal{L}(x \cdot y)$ or $uv \in \mathcal{L}(x \cdot z)$, so $w = uv \in \mathcal{L}(x \cdot y + x \cdot z)$. The other direction is equally simple. □

We collect all the equalities proved in Tables 8 and 9.

$$
\begin{array}{ll}
(a.x) \cdot y & \leftrightarrows a.(x \cdot y) \\
(x \cdot y) \cdot z & \leftrightarrows x \cdot (y \cdot z) \\
(x + y) \cdot z & \leftrightarrows x \cdot z + y \cdot z \\
\mathbf{1} \cdot x & \leftrightarrows x \\
x \cdot \mathbf{1} & \leftrightarrows x \\
\mathbf{0} \cdot x & \leftrightarrows \mathbf{0}
\end{array}
$$

Table 8: Bisimulation laws for Sequential Algebra ($a \in \mathcal{A}$).

$$
\begin{array}{ll}
x \cdot \mathbf{0} & \approx \mathbf{0} \\
x \cdot (y + z) & \approx x \cdot y + x \cdot z
\end{array}
$$

Table 9: Language equivalence laws for Sequential Algebra ($a \in \mathcal{A}$).

Again, we can prove both language equivalence and bisimulation equivalence are a congruence relation on SA terms.

Exercises

3.1.1 Use the operational rules to find automata for the following sequential expressions. In each case, use the bisimulation laws to simplify the resulting automata.

(a) $a.\mathbf{1} \cdot b.\mathbf{1}$;

(b) $(a.\mathbf{0} + b.\mathbf{1}) \cdot (a.\mathbf{0} + b.\mathbf{1})$;

(c) $(\mathbf{1} + a.\mathbf{1}) \cdot (\mathbf{1} + a.\mathbf{1})$.

3.1.2 Prove language equivalence and bisimulation equivalence are congruence relations on SA terms.

3.1.3 Suppose language L is regular. Show also L^2, defined by $L^2 = L \cdot L$, is regular. Similarly, show $L^3, L^4, \ldots$ are regular.

3.2 Iteration

We have presented an automaton as a system of recursive equations over a set of added names, a set of unknowns. If we add one more operator to the language, we can express the automaton without these added names, we can *solve* the equations. Over the sequential algebra SA, this is in general not possible, but it becomes possible if we add iteration.

Definition 3.7. The Iteration Algebra IA extends SA with the unary operator $\_^*$ denoting *iteration*. Terms over IA are called *iteration expressions* or *regular expressions*.

The operational rules for iteration are given in Table 10.

$$\frac{x \xrightarrow{a} x'}{x^* \xrightarrow{a} x' \cdot x^*} \qquad \frac{}{x^* \downarrow}$$

Table 10: Additional operational rules for IA ($a \in \mathcal{A}$).

Notice that the rule for iteration presupposes the presence of sequential composition, we have IA only as an extension of SA. The rules for iteration state that the iteration can be entered by executing a step from the inside, from the body; then after completing the body, the iteration can be entered again. Alternatively, iteration can be exited, as every iteration state is a final state.

We see that the states reachable from x^* are all states $x' \cdot x^*$, where x' is reachable from x (except maybe x itself). Notice that in this case, it does not hold any longer that every reachable state is a subterm of the starting term. Still, we can conclude that for each iteration expression, the set of reachable states is finite. This implies the following theorem.

Theorem 3.8. Let x denote a finite automaton. Then x^* denotes a finite automaton.

As a consequence, every term over IA without added names has a finite automaton. Further on, we find the reverse is also true: every finite automaton is language equivalent to the automaton of a term over IA without added names.

Another consequence is that whenever x denotes a regular process, also x^* denotes a regular process.

We can determine the languages that are accepted by iteration expressions.

Theorem 3.9. Let x be an iteration expression. $\mathcal{L}(x^*) = \{u_1 u_2 \ldots u_n \mid n \geq 0, u_i \in \mathcal{L}(x) \text{ for } i = 1, \ldots, n\}$.

Proof. Straightforward. □

If $n = 0$, expression $u_1 u_2 \dots u_n$ denotes the empty string.
Now we investigate laws for iteration, preserving bisimulation.

Theorem 3.10. The following laws hold for all terms over IA, possibly containing added names.

1. $\mathbf{0}^* \leftrightarrows \mathbf{1}$,
2. $x^* \leftrightarrows x \cdot x^* + \mathbf{1}$,
3. $(x + \mathbf{1})^* \leftrightarrows x^*$.

Proof.
1. $\mathbf{0}^* \leftrightarrows \mathbf{1}$. Only the last rule of Table 10 applies.
2. $x^* \leftrightarrows x \cdot x^* + \mathbf{1}$. The automaton of x^* is exactly the same as the automaton of $x \cdot x^* + \mathbf{1}$, apart from the name of the initial state, and so the automata are bisimilar. The set of reachable states are the states $x' \cdot x^*$, for each x' reachable from x except x itself. The initial state is a final state.
3. $(x + \mathbf{1})^* \leftrightarrows x^*$. If the body of the iteration is a final state, this is ignored. Only steps of the body count, as the left rule in Table 10 shows. The two automata are exactly the same, apart from a bisimulation relation that links the initial states and links each state $x' \cdot (x + \mathbf{1})^*$ to $x' \cdot x^*$.

□

We collect the equalities proved in Table 11.

x^*	$\leftrightarrows x \cdot x^* + \mathbf{1}$
$(x + \mathbf{1})^*$	$\leftrightarrows x^*$
$\mathbf{0}^*$	$\leftrightarrows \mathbf{1}$

Table 11: Bisimulation laws for Iteration Algebra ($a \in \mathcal{A}$).

Again, we can prove both language equivalence and bisimulation equivalence are a congruence relation on IA terms.

We can also define the iteration of a language: $L^* = \{u_1 u_2 \dots u_n \mid u_i \in L\}$, containing any concatenation of strings in L. If $n = 0$, $u_1 u_2 \dots u_n$ denotes the empty string. Now if x is any expression over IA, then we can establish $(\mathcal{L}(x))^* = \mathcal{L}(x^*)$. As a corollary, we have the following proposition.

Proposition 3.11. Let L be a regular language. Then also L^* is regular.

Example 3.12. $L = \{a^n b^k \mid n \neq k\}$ is not regular. For, if L would be regular, then also

$$\{a^n b^n \mid n \geq 0\} = \mathcal{L}((a\mathbf{1})^* \cdot (b\mathbf{1})^*) \cap \overline{L}$$

would be regular, and that is a contradiction.

Theorem 3.13. Let x, y, z be three terms over IA. Suppose $x \leftrightarrows y \cdot x + z$, and suppose $y \downarrow$ does not hold. Then $x \leftrightarrows y^* \cdot z$.

Proof. We compare the set of reachable states. Since $x \leftrightarrows y \cdot x + z$ and $y \not\downarrow$ (the initial state of y is not a final state), every initial step of x must correspond to an initial step of y, leading to a state $y' \cdot x$ or an initial step of z, leading to a state z'. Thus, the reachable states of x can be named as x for the initial state, states $y' \cdot x$, for every y' reachable from y except y itself, and states z' for every z' reachable from z except z itself.

On the other hand, the reachable states of $y^* \cdot z$ are $y^* \cdot z$ for the initial state, states $y' \cdot (y^* \cdot z)$ for every y' reachable from y except y itself, and states z' for every z' reachable from z except z itself.

Finally, the initial state is a final state just in case the initial state of z is a final state. □

This proof principle is the key in solving a recursive specification over MA to an expression over IA. The extra condition is really necessary, as the following example shows.

Example 3.14. Take $p = (a.\mathbf{1} + \mathbf{1})^* \cdot b.\mathbf{1}$ and $q = (a.\mathbf{1})^* \cdot (b.\mathbf{1} + c.\mathbf{1})$. These two expressions are different, do not accept the same language, as q has a c-step from the initial state to a final state, and p does not. Now we calculate:

$$q = (a.\mathbf{1})^* \cdot (b.\mathbf{1} + c.\mathbf{1}) \leftrightarrows (a.\mathbf{1}) \cdot ((a.\mathbf{1})^* \cdot (b.\mathbf{1} + c.\mathbf{1})) + b.\mathbf{1} + c.\mathbf{1} \leftrightarrows a.q + b.\mathbf{1} + c.\mathbf{1} \leftrightarrows$$

$$\leftrightarrows a.q + a.q + b.\mathbf{1} + b.\mathbf{1} + c.\mathbf{1} \leftrightarrows a.q + (a.q + b.\mathbf{1} + c.\mathbf{1}) + b.\mathbf{1} \leftrightarrows a.q + q + b.\mathbf{1} \leftrightarrows$$

$$\leftrightarrows (a.\mathbf{1} + \mathbf{1}) \cdot q + b.\mathbf{1}.$$

Thus, q satisfies the other condition of Theorem 3.13, but not the conclusion.

There is a variant of the previous theorem that also holds.

Theorem 3.15. Let x, y, z be three terms over IA. Suppose $x \approx y \cdot x + z$, and suppose $y \downarrow$ does not hold. Then $x \approx y^* \cdot z$.

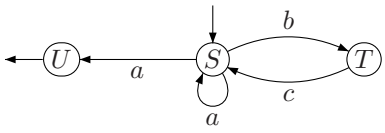

Figure 3.2: Example solving equations.

Example 3.16. Consider the simple automaton in Figure 3.2. A linear recursive specification is

$$\begin{aligned} S &= a.S + b.T + a.U \\ T &= c.S \\ U &= \mathbf{1}. \end{aligned}$$

U is already 'solved', written without added names. Once we have solved S, we have also solved T, as T is expressed in terms of S. If we want to solve for S, we can write

$$S \leftrightarrows a.S + b.T + a.U \leftrightarrows a.S + b.c.S + a.\mathbf{1}.$$

Now collect all terms containing S on the right-hand side, take S 'outside the brackets' on the right-hand side. For this, we can use sequential composition. We can write

$$S \leftrightarrows a.S + b.c.S + a.\mathbf{1} \leftrightarrows a.\mathbf{1} \cdot S + b.c.\mathbf{1} \cdot S + a.\mathbf{1} \leftrightarrows (a.\mathbf{1} + b.c.\mathbf{1}) \cdot S + a.\mathbf{1},$$

and we have collected the terms containing S.

Thus, the behaviour of S is as follows. From the start, the behaviour $a.\mathbf{1} + b.c.\mathbf{1}$ can be executed any number of times, can be iterated, until at some point the 'exit' $a.\mathbf{1}$ is chosen. By Theorem 3.13, we can write

$$S \leftrightarrows (a.\mathbf{1} + b.c.\mathbf{1}) \cdot S + a.\mathbf{1} \leftrightarrows (a.\mathbf{1} + b.c.\mathbf{1})^* \cdot a.\mathbf{1},$$

thereby solving S. Using this, we can also solve T:

$$T \leftrightarrows c.(a.\mathbf{1} + b.c.\mathbf{1})^* \cdot a.\mathbf{1}.$$

In the derivation above, the symbol $\leftrightarrows$ was used. When solving for a variable in other cases, the distributive law is necessary, so not bisimulation equivalence but language equivalence must be used. Consider the following example.

Example 3.17. Consider Figure 3.3. The recursive specification becomes:

$$\begin{aligned} S &= a.T + \mathbf{1} \\ T &= b.S + \mathbf{1} \end{aligned}$$

Solving these equations goes as follows:

$$\begin{aligned} S &\approx a.T + \mathbf{1} \approx a.(b.S + \mathbf{1}) + \mathbf{1} \approx a.b.S + a.\mathbf{1} + \mathbf{1} \approx (a.b.\mathbf{1})^* \cdot (a.\mathbf{1} + \mathbf{1}) \\ T &\approx b.S + \mathbf{1} \approx b.(a.T + \mathbf{1}) + \mathbf{1} \approx b.a.T + b.\mathbf{1} + \mathbf{1} \approx (b.a.\mathbf{1})^* \cdot (b.\mathbf{1} + \mathbf{1}). \end{aligned}$$

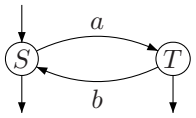

Figure 3.3: Another example solving equations.

We can do this procedure in general. This is formulated in the following theorem.

Theorem 3.18. Let L be a regular language. Then there is a term x over IA without added variables with $L = \mathcal{L}(x)$.

Proof. If L is a regular language, then there is an automaton M with $\mathcal{L}(M) = L$. Take a linear recursive specification representing M. Then, use the laws of Table 11 and Theorem 3.15 to solve each variable of this specification. These calculations preserve language equivalence.

In order to remove a certain unknown P from the right-hand side of the equations, the equation of P needs to be written without P. If P does not

contain a summand of the form $a.P$, this is immediate, otherwise Theorem 3.15 must be used. The result is then substituted in all other equations, and the number of added names is reduced by one. The resulting equations may not be linear any longer, but all right-hand sides can be reduced so that each added name occurs at most once, preceded by a term of which the initial state is not a final state.

This procedure can be repeated, until an expression without variables is reached. □

Example 3.19. We provide another example of this procedure. Consider a specification over 3 variables where all possible steps are present.

$$\begin{aligned} S &= a.S + b.T + c.U \\ T &= d.S + e.T + f.U \\ U &= g.S + h.T + i.U + \mathbf{1} \end{aligned}$$

In order to solve for S, calculate as follows:

$$S \approx aS + bT + cU \approx a\mathbf{1} \cdot S + bT + cU \approx (a\mathbf{1})^* \cdot (bT + cU) \approx$$
$$\approx (a\mathbf{1})^* \cdot bT + (a\mathbf{1})^* \cdot cU,$$

and S has been eliminated. Now use this in the other two equations. They become:

$$T \approx dS + eT + fU \approx d(a\mathbf{1})^* \cdot bT + d(a\mathbf{1})^* \cdot cU + eT + fU \approx$$
$$\approx (d(a\mathbf{1})^* \cdot b\mathbf{1} + e\mathbf{1}) \cdot T + (d(a\mathbf{1})^* \cdot c\mathbf{1} + f\mathbf{1}) \cdot U$$

and

$$U \approx gS + hT + iU + \mathbf{1} \approx g(a\mathbf{1})^* \cdot bT + g(a\mathbf{1})^* \cdot cU + hT + iU + \mathbf{1} \approx$$
$$(g(a\mathbf{1})^* \cdot b\mathbf{1} + h\mathbf{1}) \cdot T + (g(a\mathbf{1})^* \cdot c\mathbf{1} + i\mathbf{1}) \cdot U + \mathbf{1}.$$

Next, eliminate T from the equation of T:

$$T \approx (d(a\mathbf{1})^* \cdot b\mathbf{1} + e\mathbf{1})^* \cdot (d(a\mathbf{1})^* \cdot c\mathbf{1} + f\mathbf{1}) \cdot U,$$

and substitute this in the equation of U:

$$U \approx (g(a\mathbf{1})^* \cdot b\mathbf{1} + h\mathbf{1}) \cdot (d(a\mathbf{1})^* \cdot b\mathbf{1} + e\mathbf{1})^* \cdot (d(a\mathbf{1})^* \cdot c\mathbf{1} + f\mathbf{1}) \cdot U +$$
$$+ (g(a\mathbf{1})^* \cdot c\mathbf{1} + i\mathbf{1}) \cdot U + \mathbf{1} \approx$$
$$\approx \{(g(a\mathbf{1})^* \cdot b\mathbf{1} + h\mathbf{1}) \cdot (d(a\mathbf{1})^* \cdot b\mathbf{1} + e\mathbf{1})^* \cdot (d(a\mathbf{1})^* \cdot c\mathbf{1} + f\mathbf{1}) + (g(a\mathbf{1})^* \cdot c\mathbf{1} + i\mathbf{1})\}^*.$$

This result can be substituted again in the equation of T to obtain an iteration expression for T. Finally, both the expressions for T and U can be substituted in the equation of S.

Usually, a lot of work can be saved by choosing the order of elimination carefully, so that the expressions do not become too long.

Combining Theorem 3.18 with Theorem 3.8, we see that the iteration expressions denote exactly the class of all automata, accepting the class of all regular languages. This is why iteration expressions are often called regular expressions.

When finding the language accepted by an iteration expression, or determining an iteration expression for a given language, it is possible to first translate to an automaton, but it is also possible to reason directly.

Example 3.20. A regular process is given by the following linear recursive specification:

$$\begin{aligned} S &= a.T + \mathbf{1} \\ T &= b.S + \mathbf{1} \end{aligned}$$

Then there is no iteration expression bisimilar to S.

Exercises

3.2.1 Use the operational rules to find automata for the following iteration expressions. In each case, use the bisimulation laws to simplify the resulting automata.

(a) $(a.\mathbf{0} + b.\mathbf{1})^*$;
(b) $\mathbf{1} \cdot (a.\mathbf{0} + b.\mathbf{1})^*$;
(c) $(a.\mathbf{1})^* \cdot (b.\mathbf{1})^*$;
(d) $(a.\mathbf{1})^* \cdot a.b.\mathbf{1}$;
(e) $(\mathbf{1} + a.\mathbf{1} + b.\mathbf{0})^*$;
(f) $(a.(b.\mathbf{1} + \mathbf{1}))^* \cdot a.b.\mathbf{1}$.

3.2.2 Find an iteration expression for the following language: $L = \{w \in \{a,b\}^* \mid \#_a(w)$ is even and $\#_b(w)$ is odd $\}$. Hint: first make an automaton with four states (odd-odd, odd-even, even-odd and even-even).

3.2.3 Find an iteration expression for each of the automata of Figures 2.3, 2.4, 2.5.

3.2.4 Find a recursive specification for $a.(a.\mathbf{1})^* \cdot (a.b.\mathbf{1} + a.\mathbf{1})^*$.

3.2.5 Prove language equivalence and bisimulation equivalence are congruence relations on IA terms.

3.2.6 Give an iteration expression for each of the following languages:

(a) $L = \{w \in \{a,b\}^* \mid$ w has a substring bba $\}$
(b) $L = \{w \in \{a,b\}^* \mid$ w has at most one pair of consecutive 0's and at most one pair of consecutive 1's $\}$
(c) $L = \{a^m b^n \mid (n+m)$ is even $\}$

3.2.7 For $\mathcal{A} = \{0,1\}$, give an iteration expression for the language of strings over $\mathcal{A}$ such that w has at least one pair of consecutive zeros. Also give a regular expression for the complement of this language.

3.2.8 Use the operational rules to draw an automaton for the iteration expression $a.(b.\mathbf{1})^*$. Also draw the automaton of the following linear recursive specification.

$$\begin{aligned} X &= a.Y + a.Z \\ Y &= b.Z + \mathbf{1} \\ Z &= b.Z + \mathbf{1} \end{aligned}$$

Construct a bisimulation between the two automata showing $X \leftrightarrows a.(b.\mathbf{1})^*$.

3.3 Interaction

In language equivalence, interaction between user and computer is modeled by the acceptance of a string by an automaton. This is a limited form of interaction. In the process model, we consider interaction between automata. In Figure 3.4, we show interacting automata.

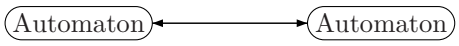

Figure 3.4: Interacting processes.

Both automata denote a regular process.

Example 3.21. Consider a relay race with two runners A and B. Their interaction is the passing of the baton: A will give the baton, denoted as $!b$ and B will take the baton, denoted $?b$. If these two actions are executed simultaneously, the baton is passed, denoted $⁉b$. A executes the process $run.!b.\mathbf{1}$ and B executes the process $?b.run.\mathbf{1}$. Together, a successful relay race is $run.⁉b.run.\mathbf{1}$.

Thus, if two regular processes are put in *parallel*, put next to each other, then they can execute actions independently, by themselves, but they can also synchronize by executing matching actions: a synchronization is the simultaneous execution of matching actions.

In general, we have as a parameter a finite set $\mathcal{D}$, called a set of *data*. For each data element $d \in \mathcal{D}$, the alphabet $\mathcal{A}$ will contain three elements:

- $?d$, receive or input data element d;
- $!d$, send or output data element d;
- $⁉d$, communication of data element d; $⁉d$ is the result of the simultaneous execution of $?d$ and $!d$.

All other elements of $\mathcal{A}$ denote actions that can execute by themselves, and cannot synchronize.

Example 3.22. We consider bounded buffers. A bounded buffer will contain elements of the finite data domain $\mathcal{D}$. The contents of the buffer can be seen as a string w over $\mathcal{D}$, where $|w| \leq k$ if $k > 0$ is the size of the buffer. Suppose, for simplicity, $\mathcal{D} = \{0, 1\}$. We have $?d$ for the input of d into the buffer, and $!d$ for the output of d from the buffer.

The following linear recursive specification has variables B_w, for each $w \in \mathcal{D}^*$ with $|w| \leq k$.

$$\begin{aligned} B_\varepsilon &= \mathbf{1}+?0.B_0+?1.B_1 \\ B_{wd} &= !d.B_w \qquad \text{if } |wd| = k \quad (w \in \mathcal{D}^*, d \in \mathcal{D}) \\ B_{wd} &= ?0.B_{0wd}+?1.B_{1wd}+!d.B_w \qquad \text{if } |wd| < k \quad (w \in \mathcal{D}^*, d \in \mathcal{D}) \end{aligned}$$

In case $k = 1$, we can derive $B_\varepsilon \leftrightarrows \mathbf{1}+?0.!0.B_\varepsilon+?1.!1.B_\varepsilon$.

In case the data set is not just bits, we get more summands in this specification. For a general finite data set $\mathcal{D}$, we use the following shorthand notation:

$$\begin{aligned} B_\varepsilon &= \mathbf{1} + \sum_{d\in\mathcal{D}} ?d.B_d \\ B_{wd} &= !d.B_w && \text{if } |wd| = k \quad (w \in \mathcal{D}^*, d \in \mathcal{D}) \\ B_{wd} &= !d.B_w + \sum_{e\in\mathcal{D}} ?e.B_{ewd} && \text{if } |wd| < k \quad (w \in \mathcal{D}^*, d \in \mathcal{D}) \end{aligned}$$

Now we look at the interaction of two buffers with capacity 1 (again $\mathcal{D} = \{0, 1\}$). The output of the first buffer will be connected to the input of the second buffer. This means the input of the first buffer is not connected (i.e., connected to the outside world), and the output of the second buffer is not connected. The input port of the first buffer is called i, the connecting port l (for link), and the output port of the second buffer is called o. See the network structure shown in Figure 3.5.

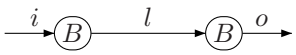

Figure 3.5: Interacting buffers.

The automata of the two buffers are shown in Figure 3.6. Each state shows the content of the buffer.

Figure 3.6: Two buffers.

The interaction will enforce the synchronisation of $l!0$ and $l?0$ to $l⁉0$, and of $l!1$ and $l?1$ to $l⁉1$. The resulting automaton is shown in Figure 3.7, where each state shows the contents of the pair of buffers.

We see that synchronisation is enforced on internal ports, but not on external ports. We will add two operators to the language: the parallel composition or merge $\|$ will allow every action separately, and will also allow synchronisation of actions, and the encapsulation operator ∂, that will disallow separate execution of actions that need to synchronise. The resulting algebra is CA, the communication algebra, the rules are presented in Table 12. CA is an extension of MA, so does not contain sequential composition or iteration.

The first two rules show the separate execution of actions by parallel components: this is called *interleaving*: the actions of x and y are interleaved or merged in time. The third rule for parallel composition says that a parallel composition can only be in a final state if both components are in a final state. The next two rules show synchronisation: matching actions by the two components can be executed simultaneously.

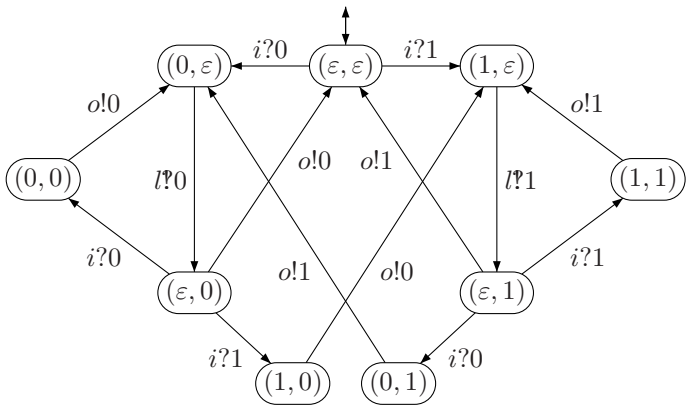

Figure 3.7: Interacting buffers.

$$\frac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y} \qquad \frac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'} \qquad \frac{x\downarrow \quad y\downarrow}{x \parallel y \downarrow}$$

$$\frac{x \xrightarrow{p?d} x' \quad y \xrightarrow{p!d} y'}{x \parallel y \xrightarrow{p\,?!\,d} x' \parallel y'} \qquad \frac{x \xrightarrow{p!d} x' \quad y \xrightarrow{p?d} y'}{x \parallel y \xrightarrow{p\,?!\,d} x' \parallel y'}$$

$$\frac{x \xrightarrow{a} x' \quad a \neq p?d, p!d}{\partial_p(x) \xrightarrow{a} \partial_p(x')} \qquad \frac{x\downarrow}{\partial_p(x)\downarrow}$$

Table 12: Operational rules for CA ($a \in \mathcal{A}$).

The rules for encapsulation ∂_p only allow a step that does not need to synchronise: thus, we will have $\partial_p(p?d.x) \leftrightarrows \mathbf{0} \leftrightarrows \partial_p(p!d.y)$ for all terms x, y.

If B^{il}_ε is the process on the left-hand side of Figure 3.6, and B^{lo}_ε is the process on the right-hand side, then we can use the operational rules to derive the process in Figure 3.7 for $\partial_l(B^{il}_\varepsilon \parallel B^{lo}_\varepsilon)$.

We have the intuition that the system of two interacting buffers of capacity one should be like a buffer of capacity two. We show the behavior of a buffer C^{io} of capacity two (with input port i and output port o) in Figure 3.8.

We see these automata are not bisimilar. They do become bisimilar if we turn the $l?0$-action into a τ, and identify the states $(0, \varepsilon)$ and $(\varepsilon, 0)$, and also turn the $l?1$-action into a τ, and identify states $(1, \varepsilon)$ and $(\varepsilon, 1)$. That is, we want to turn communicating actions at internal ports into τ, thereby *hiding* these internal communication actions, making them invisible. Indeed, it can be seen that these τ-steps are inert, and there is a branching bisimulation between the two automata.

Turning the $l?b$-actions into τ will be done by an operator $\tau_l()$. This operator is called the *abstraction* operator. Now we consider the operational rules for the abstraction operator. This is very straightforward, see Table 13.

In addition, we will use that all operational rules that hold for other operators for $a \in \mathcal{A}$, will from now on hold for $a \in \mathcal{A} \cup \{\tau\}$.

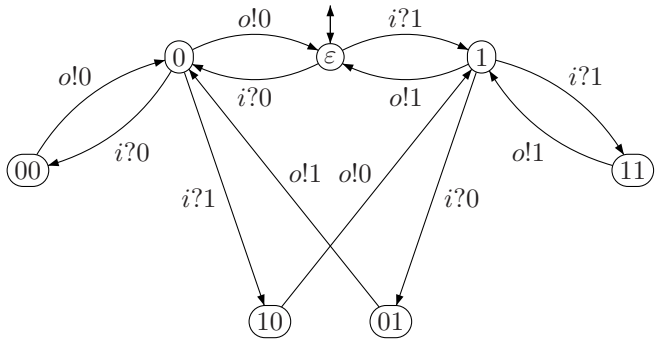

Figure 3.8: Buffer C^{io} of capacity 2.

$$\frac{x \xrightarrow{\text{?}_i d} x'}{\tau_i(x) \xrightarrow{\tau} \tau_i(x')} \qquad \frac{x \xrightarrow{a} x' \quad a \neq \text{?}_i d}{\tau_i(x) \xrightarrow{a} \tau_i(x')} \qquad \frac{x \downarrow}{\tau_i(x) \downarrow}$$

Table 13: Operational rules for abstraction ($a \in \mathcal{A} \cup \{\tau\}$).

Applying this to the interacting buffers of Figure 3.7 gives indeed that this is branching bisimilar to the buffer of capacity two of Figure 3.8, so

$$C^{io} \leftrightarrows_{\text{b}} \tau_2(\partial_2(B_\varepsilon^{12} \parallel B_\varepsilon^{23})).$$

Theorem 3.23. Let expressions x, y of CA denote a finite automaton. Then also $x \parallel y$, $\partial_p(x)$, $\tau_p(x)$ denote a finite automaton.

Again, this does not imply that recursive specifications over CA denote regular processes. A simple example is $S = \mathbf{1} + a.(S \parallel b.\mathbf{1})$. Using the operational rules, it can be seen that (after reduction) this specification denotes the nonregular process of Figure 2.9. On the other hand, any recursive specification over MA using the operators $\partial_p()$ or $\tau_p()$ will denote a regular process. We can also define these operators on languages, but will not do so as we have no use for that.

Next, we will investigate laws for the operators of CA. Table 14 lists some simple laws for parallel composition. Notice that the fourth law can not be formulated in CA, as it uses sequential composition.

$$\begin{array}{lll}
x \parallel y & \leftrightarrows y \parallel x & \\
(x \parallel y) \parallel z & \leftrightarrows x \parallel (y \parallel z) & \\
x \parallel \mathbf{1} & \leftrightarrows x & \\
x \parallel \mathbf{0} & \leftrightarrows x \cdot \mathbf{0} & \\
p!d.x \parallel p?d.y & \leftrightarrows p\text{?}d.(x \parallel y) + p!d.(x \parallel p?d.y) + p?d.(p!d.x \parallel y) & \\
a.x \parallel b.y & \leftrightarrows a.(x \parallel b.y) + b.(a.x \parallel y) & \text{if } \{a,b\} \neq \{p!d, p?d\}
\end{array}$$

Table 14: Bisimulation laws of parallel composition.

The main difficulty in developing further laws lies in the fact that it is true that $(x+y) \parallel z \approx x \parallel z+y \parallel z$, but it is *not* true that $(x+y) \parallel z \leftrightarrows x \parallel z+y \parallel z$. This can be easily seen using the operational rules (no communication in this example):

$$(a\mathbf{1} + b\mathbf{1}) \parallel c\mathbf{1} \leftrightarrows ac\mathbf{1} + bc\mathbf{1} + c(a\mathbf{1} + b\mathbf{1})$$

Notice this equation cannot be inferred from the axioms in Table 14. On the other hand,

$$a\mathbf{1} \parallel c\mathbf{1} + b\mathbf{1} \parallel c\mathbf{1} \leftrightarrows ac\mathbf{1} + bc\mathbf{1} + ca\mathbf{1} + cb\mathbf{1}$$

Equating these would amount to adopting the non-valid distributive law.

Next, we consider laws for encapsulation, in Table 15. Using these laws, we can eliminate an occurrence of encapsulation in a recursive specification over MA (adding extra variables for the encapulation of a variable). For example, an equation $S = \partial_p(a.S + p?d.S) + b.S + p!d.S$ can be converted to the specification

$$\begin{aligned} S &= a.T + b.S + p!d.S \\ T &= a.T + b.T \end{aligned}$$

$$\begin{array}{lll} \partial_p(\mathbf{0}) & \leftrightarrows \mathbf{0} & \\ \partial_p(\mathbf{1}) & \leftrightarrows \mathbf{1} & \\ \partial_p(a.x) & \leftrightarrows \mathbf{0} & \text{if } a = p!d, p?d \\ \partial_p(a.x) & \leftrightarrows a.\partial_p(x) & \text{otherwise} \\ \partial_p(x+y) & \leftrightarrows \partial_p(x) + \partial_p(y) & \\ \partial_p(\partial_p(x)) & \leftrightarrows \partial_p(x) & \\ \partial_p(\partial_q(x)) & \leftrightarrows \partial_q(\partial_p(x)) & \end{array}$$

Table 15: Bisimulation laws of encapsulation.

Laws for abstraction are similar, see Table 16. Using the laws, an occurrence of abstraction in an MA recursive specification can be eliminated.

$$\begin{array}{lll} \tau_p(\mathbf{0}) & \leftrightarrows \mathbf{0} & \\ \tau_p(\mathbf{1}) & \leftrightarrows \mathbf{1} & \\ \tau_p(p?d.x) & \leftrightarrows \tau.\tau_p(x) & \\ \tau_p(a.x) & \leftrightarrows a.\tau_p(x) & \text{otherwise} \\ \tau_p(x+y) & \leftrightarrows \tau_p(x) + \tau_p(y) & \\ \tau_p(\tau_p(x)) & \leftrightarrows \tau_p(x) & \\ \tau_p(\tau_q(x)) & \leftrightarrows \tau_q(\tau_p(x)) & \end{array}$$

Table 16: Bisimulation laws of abstraction.

Exercises

3.3.1 Give an automaton for the following processes by means of the operational rules. Next, reduce the resulting automaton by bisimulation.

(a) $a.\mathbf{1} \parallel b.\mathbf{1}$

(b) $a.b.\mathbf{1} \parallel c.\mathbf{1}$

(c) $(a.\mathbf{1} + b.\mathbf{1}) \parallel c.\mathbf{1}$

(d) $a.\mathbf{1} \parallel \mathbf{0}$

3.3.2 Give an automaton for the following processes. Laws and bisimulation may be used to reduce the size of the automaton generated by the operational rules.

(a) $p!d.p!d.\mathbf{1} \parallel p?d.p?d.\mathbf{1}$

(b) $\partial_p(p!d.p!d.\mathbf{1} \parallel p?d.p?d.\mathbf{1})$

(c) $(p!d.\mathbf{1} + p?d.\mathbf{1}) \parallel (p!d.\mathbf{1} + p?d.\mathbf{1})$

(d) $\partial_p((p!d.\mathbf{1} + p?d.\mathbf{1}) \parallel (p!d.\mathbf{1} + p?d.\mathbf{1}))$

(e) $p!d.p?d.\mathbf{0} \parallel (p!d.\mathbf{0} + p?d.\mathbf{0})$

(f) $p!d.p?d.\mathbf{1} \parallel (p!d.(a.\mathbf{1} + \mathbf{1}) + p?d.\mathbf{1})$

(g) $\partial_p((a.p!d.b.\mathbf{1})^* \parallel p?d.\mathbf{1})$

3.3.3 Establish whether the property $\partial_p(x \parallel y) \leftrightarrows \partial_p(x) \parallel \partial_p(y)$ holds for arbitrary processes x and y and port p.

3.3.4 Provide a recursive specification for a stack with input port i and output port o. Assume that the stack can contain elements from the (finite) set $\mathcal{D}$.

3.3.5 Give an automaton for the following processes by means of the operational rules:

(a) $a.(p!d.\mathbf{1})^* \parallel b.p?d.\mathbf{1}$

(b) $a.p!d.\mathbf{1} \parallel b.(p?d.\mathbf{1})^*$

(c) $a.(p!d.\mathbf{1})^* \parallel b.(p?d.\mathbf{1})^*$

3.3.6 Give an example of a process x such that x does not have a deadlock but $\partial_p(x)$ does. Also, give an example of a process y such that y has a deadlock but $\partial_p(y)$ does not.

3.3.7 Give the automaton of a bounded buffer of capacity 1 in case $\mathcal{D} = \{0, 1, 2\}$.

3.3.8 Adapt the definition of the bounded buffer in Example 3.22, so that it can still receive an input when it is full, leading to an error state. Nothing further can happen in this error state: it is a deadlock state. Now compose two of these unreliable buffers of capacity 1 in case $\mathcal{D} = \{0, 1\}$. Draw the resulting automaton.

3.3.9 Figure 3.7 shows the automaton of the process $\partial_l(B_\varepsilon^{il} \parallel B_\varepsilon^{lo})$. Now draw the automaton of the process $B_\varepsilon^{il} \parallel B_\varepsilon^{lo}$.

3.3.10 Verify that the interacting buffers of Figure 3.7 is branching bisimilar to the buffer of capacity two of Figure 3.8, so

$$\tau_l(\partial_l(B_\varepsilon^{il} \parallel B_\varepsilon^{lo})) \leftrightarrows_{\mathrm{b}} C^{io}$$

by constructing a branching bisimulation.

3.3.11 Give an automaton for the following processes by means of the operational rules:

(a) $\tau_p(\partial_p(p!d.p!d.\mathbf{1} \parallel p?d.p?d.\mathbf{1}))$
(b) $\tau_p(\partial_p((p!d.\mathbf{1} + p?d.\mathbf{1}) \parallel (p!d.\mathbf{1} + p?d.\mathbf{1})))$
(c) $\tau_p(p!d.p?d.\mathbf{0} \parallel (p!d.\mathbf{0} + p?d.\mathbf{0}))$
(d) $\tau_p(p!d.p?d.\mathbf{1} \parallel (p!d.(a.\mathbf{1} + \mathbf{1}) + p?d.\mathbf{1}))$
(e) $\tau_p(\partial_p((a.p!d.b.\mathbf{1})^* \parallel p?d.\mathbf{1}))$

Then give branching bisimilar automata with a minimal number of states for each of these.

3.3.12 Generalize the encapsulation and abstraction operators to $\partial_H()$, $\tau_H()$ for any set of ports H. Provide operational rules and laws.

3.4 Mutual exclusion protocol

We consider a simple mutual exclusion protocol. A mutual exclusion protocol concerns the exclusive access by components of a system to a shared resource while using that shared resource. A component is in its critical section when it is using the shared resource. We consider a protocol due to Peterson. The protocol should guarantee that at most one component of a system is in its critical section.

The formulation of the protocol usually uses shared variables in order to achieve coordination of components. We have to translate this to the communication paradigm we have adopted here. In order to do this, we model the shared variables as separate processes.

The protocol uses three shared variables C, D, T and two components P, Q. The variables C, D are boolean variables, T (turn) has value p or q. The value of T is the component that last started an attempt to enter its critical section. If C is false, this signifies P is not in its critical section, likewise D is false means Q is not in its critical section. If P intends to enter its critical section, it must assign the value true to C before it checks the value of D, to prevent situations in which the value of both variables is true. Similarly for Q. Still, situations can arise in which the value of both C and D is true. In order to prevent deadlock in this case, each component checks whether the other one last started an attempt to enter its critical section, and the one of which this check actually succeeds actually enters its critical section.

The variables are modeled as follows:

$$\begin{aligned}
C &= \mathbf{1} + c!f.C + c?f.C + c?t.\overline{C} \\
\overline{C} &= c!t.\overline{C} + c?t.\overline{C} + c?f.C \\
D &= \mathbf{1} + d!f.D + d?f.D + d?t.\overline{D} \\
\overline{D} &= d!t.\overline{D} + d?t.\overline{D} + d?f.D \\
T &= \mathbf{1} + t!p.T + t?p.T + t?q.\overline{T} \\
\overline{T} &= t!q.\overline{T} + t?q.\overline{T} + t?p.T
\end{aligned}$$

In each case, the automaton has two states. For P, Q, we draw the automata in Figure 3.9.

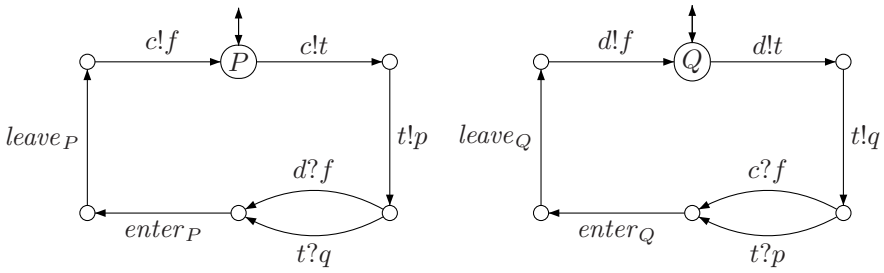

Figure 3.9: Components P, Q in mutual exclusion protocol.

Putting the whole thing together results in the automaton

$$\partial_{c,d,t}(P \parallel Q \parallel C \parallel D \parallel T)$$

in Figure 3.10. In each state, we put the values of the variables C, D, T (in this order).

The conclusion is, that the two components cannot be in the critical section at the same time: if one component executes an *enter*-action, then it needs to execute a *leave*-action before the other component can execute an *enter*-action.

Again, all actions from the set $\{c⁉b, d⁉b, t⁉r \mid b \in \{t, f\}, r \in \{p, q\}\}$ can be renamed into τ. In this case, branching bisimulation will not remove all τ-steps. The result is shown in Figure 3.11.

Exercises

3.4.1 Define three processes that repeatedly perform some task ($i = 1, 2, 3$):

$$P^i = \mathbf{1} + i?\mathit{begin}.i!\mathit{end}.P^i.$$

Now define a scheduler process S that synchronizes with all actions of the three processes, and makes sure that the processes begin in order, so first P^1 must begin, then P^2, P^3 and then P^1 again. The order of ending is arbitrary. Draw the transition system of $\partial_{1,2,3}(S \parallel P^1 \parallel P^2 \parallel P^3)$, and verify the actions $i⁉\mathit{begin}$ occur in the correct order. Also verify that for any subset of $\{1, 2, 3\}$, there is a state of the system where that subset of processes is active.

3.4.2 Verify that the mutual exclusion protocol of Figure 3.10 is, after abstraction, branching bisimilar to the process in Figure 3.11, by constructing a branching bisimulation. Verify that none of the τ-steps in Figure 3.11 is inert.

3.5 Alternating bit protocol

In this section, we have a look at a communication protocol. This protocol is often referred to as the Alternating-Bit Protocol in the literature. A communication protocol concerns the transmission of data through an unreliable channel in such a way that – despite the unreliability – no information will get lost. The communication network used in the example is shown in Figure 3.12.

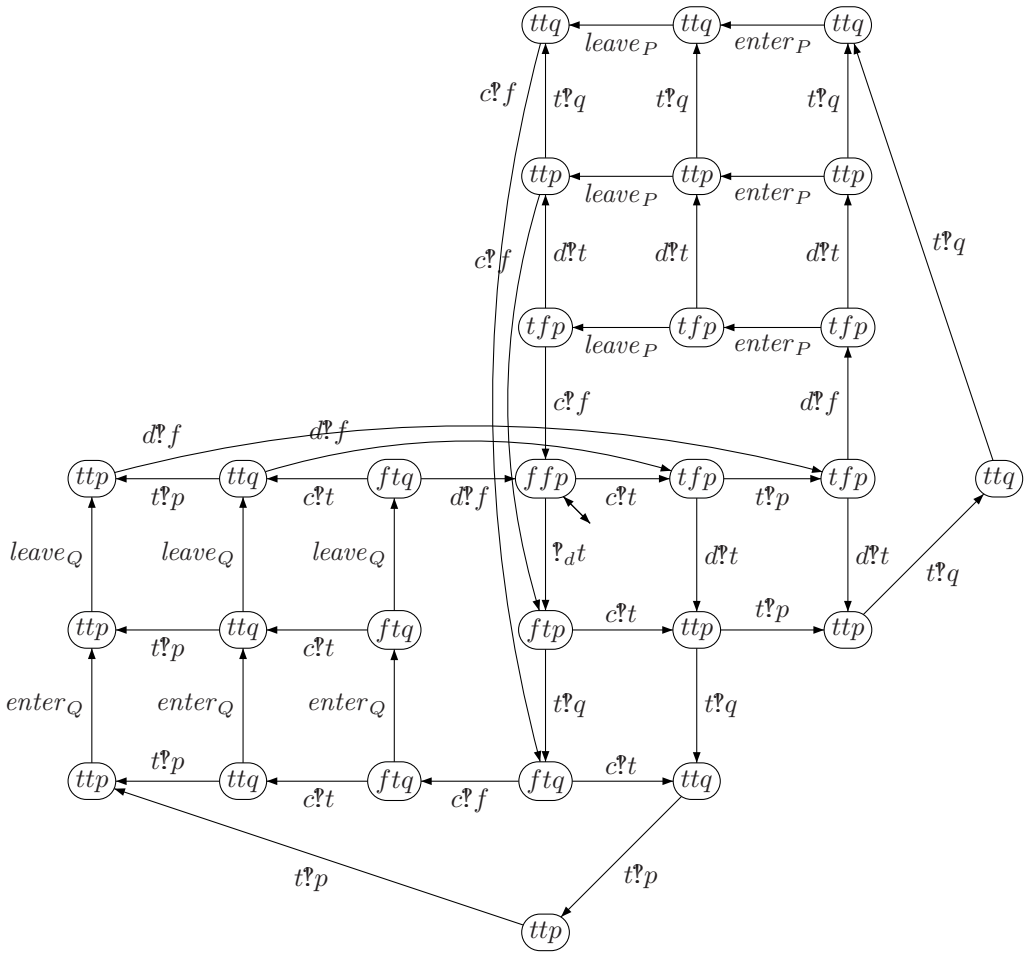

Figure 3.10: Mutual exclusion protocol $\partial_{c,d,t}(P \parallel Q \parallel C \parallel D \parallel T)$.

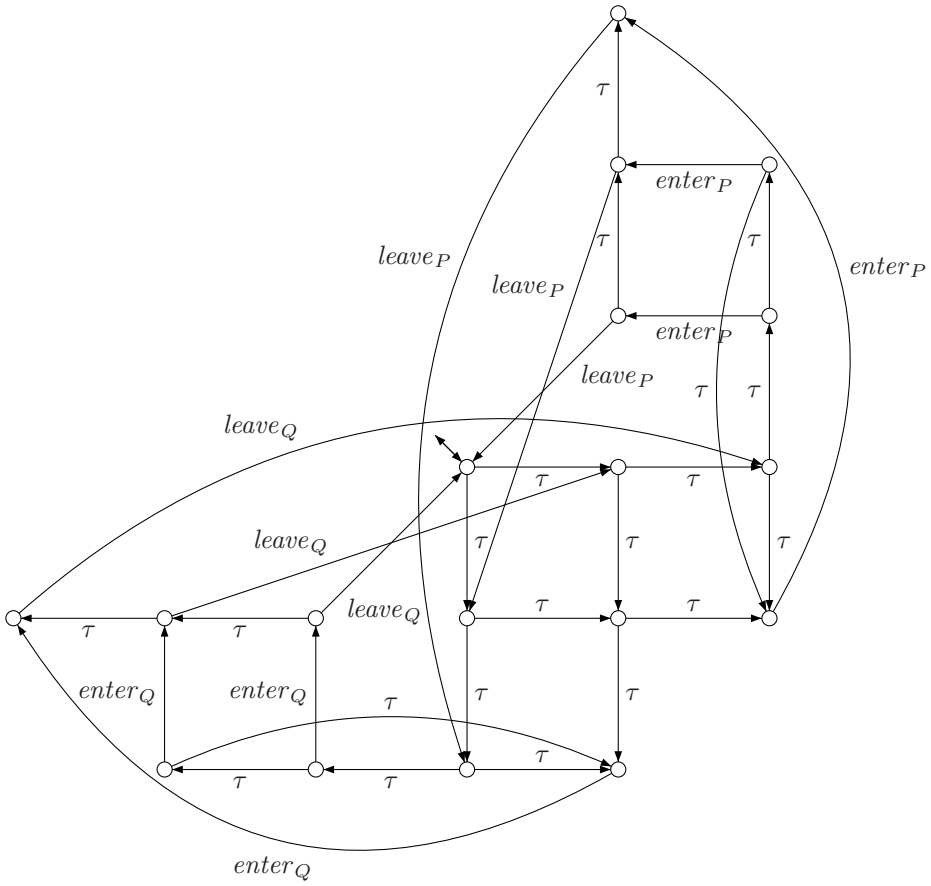

Figure 3.11: Mutual exclusion protocol $\tau_{c,d,t}(\partial_{c,d,t}(P \parallel Q \parallel C \parallel D \parallel T))$, after reduction.

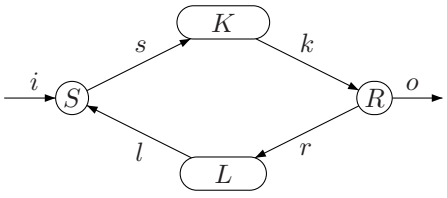

Figure 3.12: Configuration of the ABP.

The following describes the components of this network. In Figure 3.12, S is the sender, sending data elements $d \in \mathcal{D}$ to the receiver R via the unreliable channel K. After having received a certain data element, R will send an acknowledgement to S via channel L which is unreliable as well (in practice, K and L are usually physically the same medium). The problem now is to define processes S and R such that no information will get lost; that is, the behavior of the entire process, apart from the communications at the internal ports s, k, l, and r, satisfies the equation of the buffer of capacity 1

$$B_\varepsilon^{io} = \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.o!d.B_\varepsilon^{io}.$$

A solution can be formulated as follows. The sender S reads a datum d at port i and passes on a sequence $d0, d0, d0, \ldots$ of copies of this datum with an appended bit 0 to K until an acknowledgement 0 is received at port l. Then, the next datum is read, and sent on together with a bit 1; the acknowledgement then is the reception of a 1. The following data element has, in turn, 0 as an appended bit. Thus, 0 and 1 form the alternating bit.

The process K denotes the data transmission channel, passing on frames of the form $d0, d1$. K may corrupt data, however, passing on $\perp$ (an error message; thus, it is assumed that the incorrect transmission of d can be recognized, for instance, using a *checksum*).

The receiver R gets frames $d0, d1$ from K, sending on d to port o (if this was not already done earlier), and the acknowledgement 0 resp. 1 is sent to L.

The process L is the acknowledgement transmission channel, and passes bits 0 or 1, received from R, on to S. L is also unreliable, and may send on $\perp$ instead of 0 or 1.

The processes S, K, R, and L can be specified by means of automata or by means of recursive specifications. Let $\mathcal{D}$ be a finite data set, define the set of frames by $\mathcal{F} = \{d0, d1 \mid d \in D\}$, and let τ be the internal action. The channels K and L are given by the following equations.

$$\begin{aligned} K &= \mathbf{1} + \sum_{x \in \mathcal{F}} s?x.(\tau.k!x.K + \tau.k!\perp.K) \\ L &= \mathbf{1} + \sum_{b=0,1} r?b.(\tau.l!b.L + \tau.l!\perp.L) \end{aligned}$$

The action τ serves to make the choice non-deterministic: the decision whether or not the frame will be corrupted is internal to the channel, and cannot be influenced by the environment. In branching bisimulation, the treatment of τ will entail that these specifications are *not* bisimilar to the ones where the τ-steps are just removed.

The sender S and the receiver R are given by the following recursive specifications ($b = 0, 1$ and $d \in \mathcal{D}$). We write $\overline{b}$ for $1 - b$.

$$\begin{aligned} S &= S_0 \\ S_b &= \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.S_{db} \\ S_{db} &= s!db.(l?\overline{b}.S_{db} + l?\perp.S_{db} + l?b.S_{\overline{b}}) \end{aligned}$$

and

$$\begin{aligned} R &= R_1 \\ R_b &= \mathbf{1} + k?\bot.r!b.R_b + \sum_{d\in\mathcal{D}} k?db.r!b.R_b + \sum_{d\in\mathcal{D}} k?d\overline{b}.o!d.r!\overline{b}.R_{\overline{b}} \end{aligned}$$

Now an expression for the whole system is

$$\partial_{s,k,l,r}(S \parallel K \parallel L \parallel R),$$

where $\partial_{s,k,l,r}$ will block all sends and receives at ports s, k, l, r, enforcing synchronisation at these ports.

Now by means of the operational rules we can derive the automaton in Figure 3.13 for the complete system. In the first half of the automaton, we use a data-element d, in the second half e. The automaton should show a branching over $\mathcal{D}$ at the exit points.

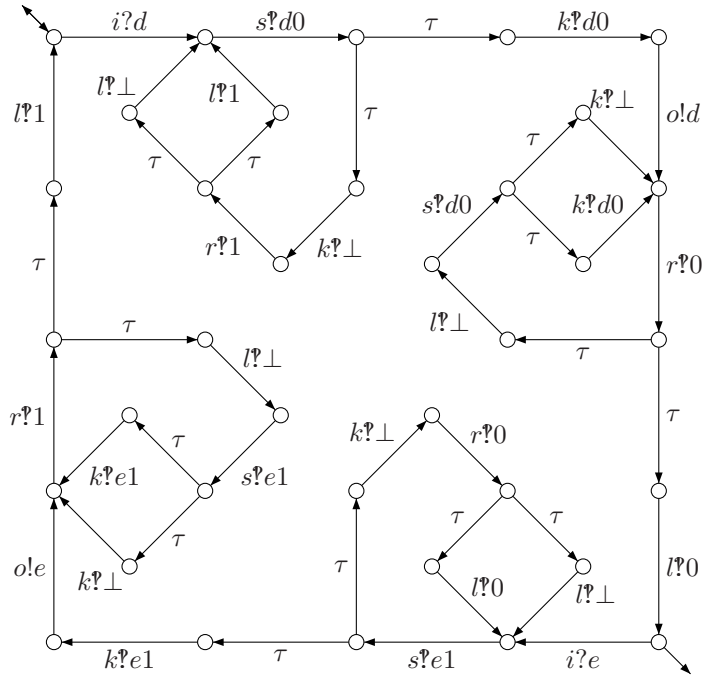

Figure 3.13: Automaton of the ABP $\partial_{s,k,l,r}(S \parallel K \parallel L \parallel R)$.

Next, we want to hide the communications at the internal ports, we want to turn communications at ports s, k, l, r into τ. Indeed, all τ-steps can be removed, and we can show branching bisimilarity with the buffer of capacity one:

$$B_\varepsilon^{io} \leftrightarrows_{\mathrm{b}} \tau_{s,k,l,r}(\partial_{s,k,l,r}(S \parallel K \parallel L \parallel R)).$$

Notice that the notion of branching bisimilarity embodies a form of *fairness*: after a number of failures of correct communication, eventually a successful communication will take place, the channel cannot be totally defective.

Exercises

3.5.1 Suppose a sender process is given by

$$S = \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.l!d.l?ack.S$$

(here, *ack* is an acknowledgement). Consider three different receiver processes:

$$\begin{aligned} R^1 &= \mathbf{1} + \sum_{d \in \mathcal{D}} l?d.o!d.l!ack.R^1 \\ R^2 &= \mathbf{1} + \sum_{d \in \mathcal{D}} l?d.l!ack.o!d.R^2 \\ R^3 &= \mathbf{1} + \sum_{d \in \mathcal{D}} l?d.(o!d.\mathbf{1} \parallel l!ack.\mathbf{1}) \cdot R^3 \end{aligned}$$

Draw automata for the sender and each of the receivers. Also draw automata for the three processes $\partial_l(S \parallel R^i)$ $(i = 1, 2, 3)$. Take $\mathcal{D} = \{0, 1\}$ if you find this easier.

3.5.2 In this exercise, a simple communication protocol is considered. Data (from some finite data set $\mathcal{D}$) are to be transmitted from a sender S to a receiver R through some unreliable channel K (see Figure 3.14).

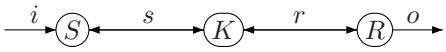

Figure 3.14: A simple communication network.

The channel may forward the data correctly, or may completely destroy data. The sender will send a data element until an acknowledgement *ack* is received. Consider the following specifications for S, K, and R:

$$\begin{aligned} S &= \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.S_d, \\ S_d &= s!d.S_d + s?ack.S \quad (\text{for all } d \in \mathcal{D}), \\ R &= \mathbf{1} + \sum_{d \in \mathcal{D}} r?d.o!d.R + r!ack.R, \\ K &= \mathbf{1} + \sum_{d \in D} s?d.(\tau.K + \tau.r!d.L), \\ L &= \mathbf{1} + r?ack.(\tau.L + \tau.s!ack.K). \end{aligned}$$

Draw the automaton of the process $\partial_{s,r}(S \parallel K \parallel R)$. Does this communication protocol behave correctly?

3.5.3 Verify that the alternating bit protocol of Figure 3.13 is branching bisimilar to the buffer of capacity one:

$$B^{io}_{\varepsilon} \leftrightarrows_{\mathrm{b}} \tau_{s,k,l,r}(\partial_{s,k,l,r}(S \parallel K \parallel L \parallel R)),$$

by constructing a branching bisimulation.

Chapter 4

Push-Down Automata

In the previous chapters, we studied finite automata, modeling computers without memory. In the next chapter, we study the general model of computers with memory. In the current chapter, we study an interesting class that is in between: a class of automata with a memory in the form of a stack, so-called *push-down automata*. We link the class of push-down automata to the class of recursive specifications over Sequential Algebra SA. The class of languages associated with push-down automata is often called the class of *context-free* languages. This class is important in the study of programming languages, in particular in parsing.

4.1 Push-down languages and processes

We consider an abstract model of computer with a memory in the form of a *stack*: this stack can be accessed only at the top: something can be added on top of the stack (push), or something can be removed from the top of the stack (pop). In Figure 4.1, we modify Figure 2.1 by adding a stack S. From the start, we allow internal moves τ, that consume no input symbol but can modify the stack.

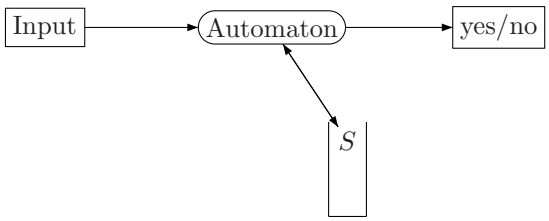

Figure 4.1: Abstract model of a push-down automaton.

Definition 4.1 (Push-down automaton)**.** A *push-down automaton* M is a six-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{B}, \rightarrow, \uparrow, \emptyset, \downarrow)$ where:

1. $\mathcal{S}$ is a finite set of states,

2. $\mathcal{A}$ is a finite input alphabet,
3. $\mathcal{D}$ is a finite data alphabet,
4. $\rightarrow \subseteq \mathcal{S} \times (\mathcal{D} \cup \{\varepsilon\}) \times (\mathcal{A} \cup \{\tau\}) \times \mathcal{D}^* \times \mathcal{S}$ is a finite set of *transitions* or *steps*,
5. $\uparrow \in \mathcal{S}$ is the initial state,
6. $\downarrow \subseteq \mathcal{S}$ is the set of final states.

If $(s, d, a, x, t) \in \rightarrow$ with $d \in \mathcal{D}$, we write $s \xrightarrow{d,a,x} t$, and this means that the machine, when it is in state s and d is the top element of the stack, can consume input symbol a, replace d by the string x and thereby move to state t. Likewise, writing $s \xrightarrow{\varepsilon,a,x} t$ means that the machine, when it is in state s and the stack is empty, can consume input symbol a, put the string x on the stack and thereby move to state t.

Example 4.2. Consider the push-down automaton in Figure 4.2. When started in the initial state with empty stack, it can either go to the final state with a τ-step, or execute a number of a's (at least one), each time putting an additional 1 on the stack. After a number of a's, the same number of b's can be executed, each time removing a 1 from the stack. When the stack is empty, a τ-transition to the final state can be taken. We see that the number of b's executed must equal the number of a's executed, and so we have the idea that this push-down automaton accepts exactly the strings $a^n b^n$, for $n \geq 0$.

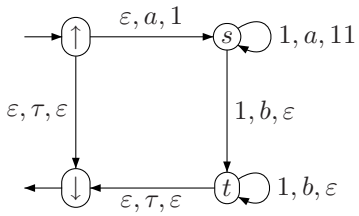

Figure 4.2: Example push-down automaton.

At any point in an execution, the state of a push-down automaton is given by a pair (s, x), where $s \in \mathcal{S}$ is the current state and $x \in \mathcal{D}^*$ is the current contents of the stack. In the initial state, the stack is empty, and in order to terminate, the stack must again be empty.

To give an example, in the push-down automaton in Figure 4.2 we can write down the following execution:

$$(\uparrow, \varepsilon) \xrightarrow{a} (s, 1) \xrightarrow{a} (s, 11) \xrightarrow{b} (t, 1) \xrightarrow{b} (t, \varepsilon) \xrightarrow{\tau} (\downarrow, \varepsilon) \downarrow .$$

We can display *all* possible executions in the following transition system, see Figure 4.3. This is the transition system that is determined by the push-down

automaton, denoting the process of the push-down automaton. Notice that the τ-step on the bottom is inert, but the τ-step on the left is not, and so this process is not branching bisimilar to the process in Figure 2.6 (but it is language equivalent).

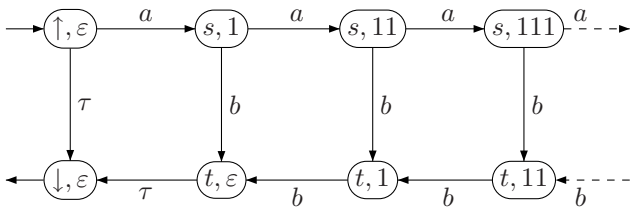

Figure 4.3: The process of the example push-down automaton.

It is not so difficult to give a push-down automaton of which the process is the counter of Figure 2.6. See Figure 4.4.

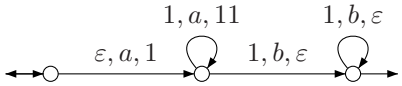

Figure 4.4: Variant of example push-down automaton.

Definition 4.3. Let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be a push-down automaton. The *transition system* or *process* of M is defined as follows:

1. The alphabet is $\mathcal{A} \cup \{\tau\}$, the set of states is $\{(s, x) \mid s \in \mathcal{S}, x \in \mathcal{D}^*\}$;
2. $(s, dy) \xrightarrow{a} (t, xy)$ iff $s \xrightarrow{d,a,x} t$, for all $y \in \mathcal{D}^*$;
3. $(s, \varepsilon) \xrightarrow{a} (t, x)$ iff $s \xrightarrow{\varepsilon,a,x} t$;
4. The initial state is $(\uparrow, \varepsilon)$;
5. $(s, \varepsilon) \downarrow$ iff $s \downarrow$.

We see that termination can only take place when the stack is empty.

Definition 4.4. Let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be a push-down automaton. The language *accepted* by M is the language of its transition system, so:

$$\mathcal{L}(M) = \{w \in \mathcal{A}^* \mid \text{for some } s \in \mathcal{S} \text{ with } s \downarrow \text{ we have } (\uparrow, \varepsilon) \xrightarrow{w} (s, \varepsilon)\}.$$

Example 4.5. Let us construct a push-down automaton for the language $\{ww^R \mid w \in \{a, b\}^*\}$. It is convenient to use a, b also as data symbols, so $\mathcal{D} = \{a, b\}$. In the initial state, a string can be read in and put on the stack. At some point (non-deterministically) it will switch to the second state, where

the stack will be read out again in reverse order (as a stack is last-in-first-out). Termination takes place when the stack is empty again. See Figure 4.5. In this Figure, z stands for an arbitrary element of $\mathcal{D} \cup \{\varepsilon\}$, so a transition $\stackrel{z,\tau,z}{\longrightarrow}$ stands for the three transitions $\stackrel{\varepsilon,\tau,\varepsilon}{\longrightarrow}$, $\stackrel{a,\tau,a}{\longrightarrow}$ and $\stackrel{b,\tau,b}{\longrightarrow}$.

A possible execution:

$$(\uparrow, \varepsilon) \stackrel{a}{\longrightarrow} (\uparrow, a) \stackrel{b}{\longrightarrow} (\uparrow, ba) \stackrel{\tau}{\longrightarrow} (\downarrow, ba) \stackrel{b}{\longrightarrow} (\downarrow, a) \stackrel{a}{\longrightarrow} (\downarrow, \varepsilon) \downarrow .$$

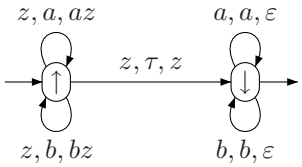

Figure 4.5: $L = \{ww^R \mid w \in \{a, b\}^*\}$.

Definition 4.6. Let $L \subseteq \mathcal{A}^*$. L is a *push-down language* iff there is a push-down automaton that accepts L.

Let T be a transition system. T is a *push-down process* iff T is branching bisimilar to the transition system of some push-down automaton.

Example 4.7. The languages $\{a^n b^n \mid n \geq 0\}$ and $\{ww^R \mid w \in \{a, b\}^*\}$ are push-down languages. They are not regular (see Example 2.60 and 2.61.

The processes of Figure 4.3, 2.17 and 2.6 are push-down processes. They are not regular (see Example 3.3).

Example 4.8. Let us consider the (last-in first-out) stack process itself. Given a finite data set $\mathcal{D}$, this process can execute the following actions:

- $i?d$, *push* data element d onto the stack (input at port i);
- $o!d$, *pop* data element d, if this is the element at the top of the stack (output at port o);
- termination can occur only when the stack is empty.

It is easy to define a push-down automaton for the stack: it has a single state which is initial and also final, and transitions $\stackrel{\varepsilon,i?d,d}{\longrightarrow}$, $\stackrel{d,i?e,ed}{\longrightarrow}$, $\stackrel{d,o!d,\varepsilon}{\longrightarrow}$ from this state to itself for all $d, e \in \mathcal{D}$.

We show the transition system of the stack in Figure 4.6 in case $\mathcal{D} = \{0, 1\}$. This is an infinite transition system. Moreover, no two states of this transition system can be (branching) bisimilar, as each state has a unique path back up to the root. Thus, the stack is a push-down process and is not regular.

Example 4.9. Consider the push-down automaton in Figure 4.7. Initially, it can stack an arbitrary number of 1's. Then, it executes exactly this number of a's before terminating. As all initial τ-steps are inert, the resulting transition system is branching bisimilar to the transition system in Figure 4.8. We conclude that this transition system is a push-down process, that shows infinite branching

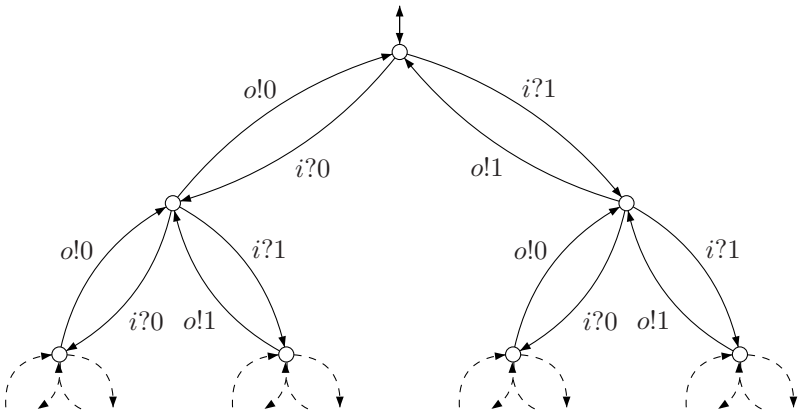

Figure 4.6: Stack over $\mathcal{D} = \{0, 1\}$.

at the root. Every state on the bottom allows a different number of a-steps to termination. This means the transition system is not (branching) bisimilar to a finite automaton, and so this is not a regular process. However, its language $\{a^n \mid n \geq 0\}$ is regular.

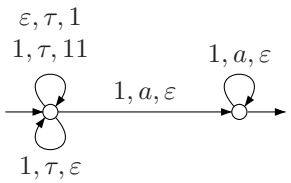

Figure 4.7: Push-down automaton that generates an infinitely branching push-down process.

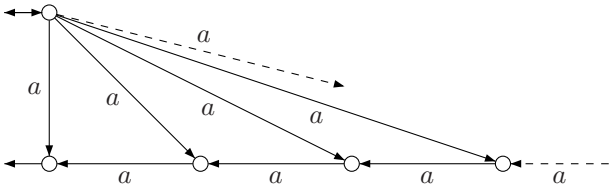

Figure 4.8: Infinitely branching process.

The definition of a push-down automaton has transitions $\xrightarrow{\varepsilon,a,x}$ and $\xrightarrow{d,a,x}$ for arbitrary sequences $x \in \mathcal{D}^*$. The notions of push-down language and push-down process do not change if we limit the set of steps to only *push* and *pop* transitions:

- A *push* transition is a transition of the form $\xrightarrow{\varepsilon,a,d}$ or $\xrightarrow{d,a,ed}$ $(d, e \in \mathcal{D})$;
- a *pop* transition is a transition of the form $\xrightarrow{d,a,\varepsilon}$ $(d \in \mathcal{D})$.

Theorem 4.10. Let L be a push-down language. Then L is also accepted by a push-down automaton using push and pop transitions only.

Let T be a push-down process. Then T is branching bisimilar to the transition system of a push-down automaton using push and pop transitions only.

Proof. We can prove the two statements at the same time. Suppose the language or the process is given by a push-down automaton using arbitrary transitions. Then we can construct a push-down automaton using only push and pop transitions for the same language or process as follows.

1. Eliminate a transition of the form $s \xrightarrow{\varepsilon,a,\varepsilon} t$ by adding a new state u, replacing the transition by the sequence of transitions $s \xrightarrow{\varepsilon,a,d} u \xrightarrow{d,\tau,\varepsilon} t$ (with d just some arbitrary element in $\mathcal{D}$).

2. Eliminate a transition of the form $s \xrightarrow{\varepsilon,a,x} t$, with $x = e_n \cdots e_1$ $(n > 1)$, by adding new states $s_2, \ldots, s_n$ and replacing the transition $s \xrightarrow{\varepsilon,a,x} t$ by the sequence of transitions

$$s \xrightarrow{\varepsilon,a,e_1} s_2 \xrightarrow{e_1,\tau,e_2e_1} \cdots \xrightarrow{e_{n-2},\tau,e_{n-1}e_{n-2}\cdots e_1} s_n \xrightarrow{e_{n-1},\tau,e_ne_{n-1}\cdots e_1} t.$$

3. Eliminate a transition of the form $s \xrightarrow{d,a,x} t$, with $x = e_n \cdots e_1$ $(n > 0, n \neq 2)$, by adding new states $s_1, \ldots, s_n$ and replacing the transition $s \xrightarrow{d,a,x} t$ by transitions $s \xrightarrow{d,a,\varepsilon} s_1$, $s_1 \xrightarrow{\varepsilon,\tau,e_1} s_2$ and $s_1 \xrightarrow{f,\tau,e_1f} s_2$ for all $f \in \mathcal{D}$, and the sequence of transitions

$$s_2 \xrightarrow{e_1,\tau,e_2e_1} \cdots \xrightarrow{e_{n-2},\tau,e_{n-1}e_{n-2}\cdots e_1} s_n \xrightarrow{e_{n-1},\tau,e_ne_{n-1}\cdots e_1} t.$$

□

Exercises

4.1.1 Construct a push-down automaton for the following languages over $\mathcal{A} = \{a, b, c\}$.

(a) $L = \{a^n b^{n+m} c^m \mid n \geq 0, m > 0\}$;
(b) $L = \{w \mid \#_a(w) < \#_b(w)\}$;
(c) $L = \{w \mid \#_a(w) + \#_b(w) = \#_c(w)\}$.

4.1.2 Give a push-down automaton for each of the following languages:

(a) $L = \{a^n b^{2n} \mid n \geq 0\}$;
(b) $L = \{a^m b^n \mid m, n \geq 0, m \neq n\}$;
(c) $L = \{a^m b^n \mid 2m = 3n + 1\}$;
(d) $L = \{a^m b^n \mid m \geq n, \quad (m - n) \mod 2 = 0\}$.

4.1.3 Give a push-down automaton for each of the following languages:

(a) $L = \{a^n b^{2n} c^m \mid n > 0, m > 0\}$;
(b) $L = \{a^n b^m c^{2m+n} \mid n > 0, m > 0\}$;

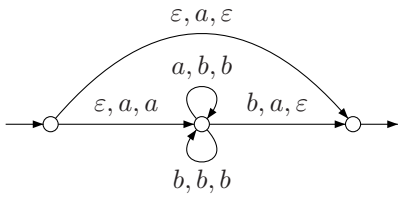

Figure 4.9: Push-down automaton for Exercise 5.

(c) $L = \{a^n b^m c^i \mid 0 \leq n + m \leq i\}$;

(d) $L = \{a^m b^i a^n \mid i = m + n\}$;

(e) $L = \{wc^n \mid w \in \{a, b\}^*, n = \#_a(w) \vee n = \#_b(w)\}$.

4.1.4 Let $L = \{a^n b^n c^n \mid n \geq 0\}$. Give a push-down automaton for the language $\overline{L}$.

4.1.5 What language is accepted by the push-down automaton shown in Figure 4.9?

4.1.6 Prove that the push-down automaton in Figure 4.5 does not accept any string not in $\{ww^R\}$.

4.1.7 Give a push-down automaton with just one state, that has the transition system of Figure 2.9 as its transition system.

4.1.8 Prove that the push-down automaton constructed in the proof of Theorem 4.10 has a transition system that is branching bisimilar to the push-down automaton started out from, by showing that all added τ-steps are inert.

4.2 Recursive specifications over SA

In Definition 3.1, we defined the Sequential Algebra SA, and provided operational rules and laws for this algebra. In Example 3.3, we already saw that by means of a recursive specification over SA, we can define certain nonregular processes. Now we consider this class, so given is a finite set of names $\mathcal{N}$ and a set of recursive equations over these names. We combine the operational rules of Tables 1 and 4 and the rules for sequential composition in Table 7, and we also combine the laws of Tables 2, 3, 8 and Table 9. Moreover, we use τ-steps.

A language that is generated by a recursive specification over SA is often called *context-free*. The origin of this name comes from the fact that the left-hand sides of the equations are single variables. These variables can be replaced by their right-hand sides, regardless of the context in which they appear.

Obviously, every recursive specification over MA is also a recursive specification over SA, so every regular language is also context-free. We will find that some languages that are not regular are context-free. Also, we find new specifications for regular languages. We will establish that every context-free language is push-down, and that every push-down language is context-free.

In the case of processes, the relationship between processes generated by recursive specifications over SA and push-down processes is not so straightforward. We will only establish a partial result.

Example 4.11. The linear specification $S = \mathbf{1} + a.S$ has the automaton with one state that is a final state, and an a-loop. Its language is $\{a^n \mid n \geq 0\}$, and the automaton denotes a regular process. Now consider the specification

$$S = \mathbf{1} + S \cdot a.\mathbf{1}.$$

By means of the operational rules, we derive the transition system in Figure 4.8: from $S \downarrow$ we infer $S \xrightarrow{a} \mathbf{1}$, from which we infer $S \cdot a.\mathbf{1} \xrightarrow{a} \mathbf{1} \cdot a.\mathbf{1}$ and so $S \xrightarrow{a} \mathbf{1} \cdot a.\mathbf{1}$ etc.. We saw this is a push-down process, not a regular process, which has a regular language.

Such infinitely branching transition systems are difficult to work with. When the set of states reachable from the initial state in one step is finite, it is easier to fix the initial part of the state space.

Example 4.12. Add one name S to SA and consider the following recursive equation:

$$S = \mathbf{1} + a.S \cdot b.\mathbf{1}.$$

We infer $S \downarrow$ and $S \xrightarrow{a} S \cdot b.\mathbf{1}$. Considering state $S \cdot b.\mathbf{1}$, we infer $S \cdot b.\mathbf{1} \xrightarrow{b} \mathbf{1}$ and $S \cdot b.\mathbf{1} \xrightarrow{a} S \cdot b.\mathbf{1} \cdot b.\mathbf{1}$. Continuing in this way, we see that we obtain exactly the transition system of Figure 2.6. This process is push-down and non-regular, and its language $\{a^n b^n \mid n \geq 0\}$ is push-down and non-regular.

The difference between the two recursive specifications in the previous examples is that in the second example, the variable S on the right-hand side is preceded by a, and in the first example, it is not preceded by any action. We say that in the second example, the variable S on the right-hand side is *guarded*. In the first example, the unguarded S on the right-hand side allows to derive a new step over and over again, thereby causing infinite branching. The example in Figure 4.7 shows that τ cannot be considered a guard, as such τ-steps might be inert and could be removed.

Definition 4.13. We call the occurrence of a variable in the right-hand side of an equation *guarded* if this occurrence is in the scope of an action-prefix operator, if this variable is preceded by an element of the alphabet $\mathcal{A}$. Thus, in the term $a.X \cdot Y + b.(X + c.Y \cdot X) + Z \cdot Z + \tau.Z$, all occurrences of X and Y are guarded, and all occurrences of Z are unguarded.

Consider a recursive specification over SA with variables $\mathcal{N}$. If Q occurs unguarded in the equation of P, we write $P \rightsquigarrow Q$. If the relation $\rightsquigarrow$ contains a cycle, we call the recursive specification *unguarded*. Otherwise, it is guarded.

The second specification in Example 4.11 is unguarded as $S \rightsquigarrow S$, the specification in Example 4.12 is guarded as S is guarded by a on the right-hand side.

In the examples we have seen so far, the right-hand side of each equation is presented as a sum of one or more summands, each of which does not contain a $+$. Thus, each right-hand side is written without brackets, as a sum of sequential terms. This format will turn out to be most useful in the sequel. Every

recursive specification can be brought into this form, by adding extra variables if necessary, and using the right distributivity over sequential composition over choice. To give an example, if we have an equation

$$X = A \cdot (B + C) \cdot D,$$

we add a new variable Y, and replace this equation by

$$\begin{aligned} X &\leftrightarrows A \cdot Y \\ Y &\leftrightarrows B \cdot D + C \cdot D. \end{aligned}$$

In the sequel, we will assume that all our recursive specifications are in this form.

Definition 4.14. A *sequential term* is a term that does not contain a $+$ operator, so it only contains action prefix, sequential composition, $\mathbf{1}$ and $\mathbf{0}$. We say a recursive specification is in *sequential form* if each right-hand side of each equation is a sum of sequential terms.

Theorem 4.15. For each recursive specification over SA, there is a bisimilar recursive specification in sequential form.

Theorem 4.16. Let the transition system M be given by a guarded recursive specification over SA. Then M is finitely branching.

Proof. Let M be given by a guarded recursive specification over SA. We can assume the specification is in sequential form. If some of the sequential terms on the right-hand side start with a variable, or a variable preceded by an inert τ, replace this variable by its right-hand side, and again use the distributive law to remove all brackets. Eventually, as the specification is guarded, this procedure must stop, and all the terms of the initial variable will start with an action. Then, all steps exiting the initial state can be read off, and we can see there are only finitely many. The resulting states are denoted by a sequential term. The procedure can be repeated for these sequential terms, and we obtain a finitely branching transition system. □

As the unguarded recursive specification in Example 4.11 has a transition system that is not bisimilar to any finitely branching transition system, it cannot be given by a guarded recursive specification.

Example 4.17. Add one name S to SA and consider the following recursive equation:

$$S = \mathbf{1} + a.S \cdot b.S.$$

This recursive specification is guarded: the first S on the right-hand side is guarded by a, the second by both a and b. The specification is also in sequential form. With the operational rules we obtain $S \downarrow$ and $S \xrightarrow{a} S \cdot b.S$. Reasoning as above, we obtain the transition system of Figure 2.9, and so this is a push-down process. Also, we have a recursive specification of the non-regular language $\{w \in \mathcal{A}^* \mid \#_a(w) = \#_b(w)$ and for all prefixes v of w we have $\#_a(v) \geq \#_b(v)\}$. As a consequence, this is also a non-regular process.

As we argued before, this process can be seen as a counter, where a denotes an increment and b a decrement. We can also interpret a as an input and b as an output, and then generalize this specification as follows:

$$S = \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.S \cdot o!d.S.$$

If the set $\mathcal{D}$ is a singleton, we just have a relabeling of the transition system of Figure 2.9, but if $\mathcal{D} = \{0, 1\}$ we get the transition system in Figure 4.10, showing the sequential form in each state of Figure 4.6. We see this is the stack. Thus, we have a guarded recursive specification of the push-down process stack.

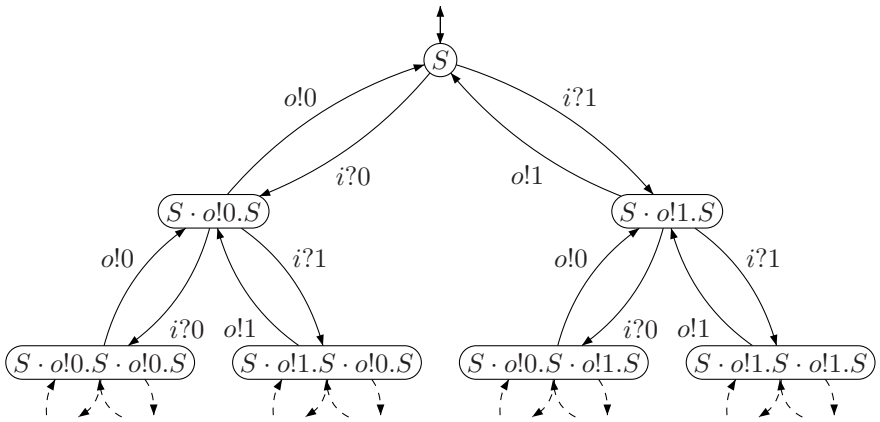

Figure 4.10: Stack over $\mathcal{D} = \{0, 1\}$.

Example 4.18. Consider the recursive equation

$$S = \mathbf{1} + a.S \cdot b.S + S \cdot S.$$

This recursive specification is unguarded, as both S's occur unguarded in the second summand on the right-hand side. Looking at the derivations, this will again yield a transition system that is infinite and has infinite branching. Nevertheless, this transition system is bisimilar to the one in the previous example, and this recursive specification does denote a context-free process, with the context-free language $\{w \in \mathcal{A}^* \mid \#_a(w) = \#_b(w)$ and for all prefixes v of w we have $\#_a(v) \geq \#_b(v)\}$.

Example 4.19. Consider the guarded recursive equation

$$S = \tau.\mathbf{1} + a.S \cdot a.\mathbf{1} + b.S \cdot b.\mathbf{1}.$$

Any a or b that is generated to the left of S will also be generated to the right of S, but the order will be reversed in the resulting string. This leads us to conclude that the language of this recursive equation is $\mathcal{L}(S) = \{ww^R \mid w \in \{a, b\}^*\}$. It is also the case that this guarded recursive equation yields a push-down process, it has the push-down automaton in Figure 4.5.

Example 4.20. Also conversely, given a context-free language, in some cases a recursive specification can be constructed. Consider the language $L = \{a^n b^m \mid n \neq m\}$. To find a recursive specification over SA for this language, we start out from the equation in Example 4.12, but now with a different variable-name:

$$T = \mathbf{1} + a.T \cdot b.\mathbf{1}.$$

Next, we have to add extra a's on the left or extra b's on the right. Variable A takes care of extra a's, variable B of extra b's. We obtain:

$$\begin{aligned} S &= A \cdot T + T \cdot B \\ T &= \mathbf{1} + a.T \cdot b.\mathbf{1} \\ A &= a.(\mathbf{1} + A) \\ B &= b.(\mathbf{1} + B) \end{aligned}$$

Notice that this recursive specification is guarded: in the first equation, variables A, B, T occur unguarded, but no variable occurs unguarded in the equations of these variables.

A process defined by a guarded recursive specification has finite branching, but the branching may be *unbounded*. We say the branching of a process p is *bounded* if there is some number N such that every state of p (in the transition system that is reduced as much as possible with respect to branching bisimulation) has fewer than N outgoing edges. Otherwise, p is said to have unbounded branching.

We give an example of a process defined by a guarded recursive specification that has unbounded branching. We note that the unboundedness is caused by the presence of a variable that has a $\mathbf{1}$ summand. We think, but are not able to prove, that this is *not* a push-down process.

Example 4.21. Consider the following guarded recursive specification:

$$\begin{aligned} S &= a.S \cdot T + b.\mathbf{1} \\ T &= \mathbf{1} + c.\mathbf{1}. \end{aligned}$$

This specification denotes a context-free process. We see $S \xrightarrow{a} S \cdot T \xrightarrow{b} \mathbf{1} \cdot T \xrightarrow{c} \mathbf{1}$. In general, we have $S \xrightarrow{a^n} S \cdot T^n \xrightarrow{b} \mathbf{1} \cdot T^n$ for every $n \geq 1$. Now $\mathbf{1} \cdot T^n \xrightarrow{c} \mathbf{1} \cdot T^k$ for every $k < n, k \geq 0$. This means state $\mathbf{1} \cdot T^n$ has n outgoing edges. Also, all the states $\mathbf{1} \cdot T^n$ are different, are not (branching) bisimilar, as $\mathbf{1} \cdot T^n$ has at most n c-steps to termination. We show the transition system of this process in Figure 4.11.

We see a process defined by a guarded recursive specification over SA can have unbounded branching. In this case, we conjecture it is not a push-down process either. In the example, this happened because there were states of the form $\mathbf{1} \cdot T^n$, where every variable in the sequence can directly contribute an outgoing step to the state.

It is not the case that every push-down process is given by a recursive specification over SA. We will give the following example.

Example 4.22. Consider the push-down automaton in Figure 4.12. There is no recursive specification over SA of which the transition system is branching

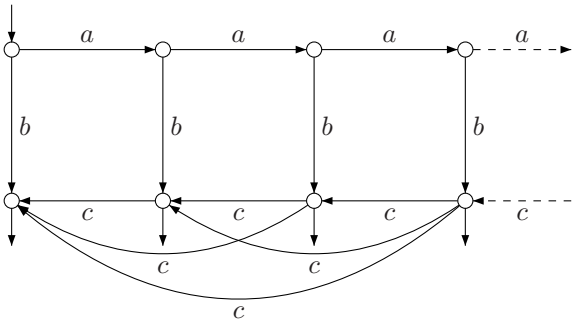

Figure 4.11: Process defined by guarded recursive specification with unbounded branching.

bisimilar to its transition system. We will not prove this fact. The reason that it has no recursive specification over SA is that the pop of a 1 can lead to two different states, a pop can take place leading to the initial state and to the final state. We say the push-down automaton is not *popchoice-free*.

We remark that a recursive specification can be found, nevertheless, if we extend the minimal algebra MA with the parallel composition operator instead of the sequential operator. This (guarded) specification is as follows:

$$S = c.\mathbf{1} + a.(S \parallel b.\mathbf{1}).$$

Using the operational rules of Table 12, we can derive a transition system. It is easy to show it is the transition system of the push-down automaton of Figure 4.12.

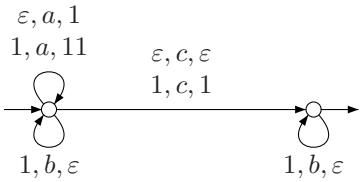

Figure 4.12: Push-down automaton that has no recursive specification over SA.

Exercises

4.2.1 Construct a push-down automaton that accepts the language generated by the specification

$$S = a.a.b.\mathbf{1} + a.S \cdot b.b.\mathbf{1}.$$

4.2.2 Construct a push-down automaton that accepts the language generated by the specification

$$S = a.b.\mathbf{1} + a.S \cdot S \cdot S.$$

4.2.3 For Example 4.11, write out the derivation of the transition system in Figure 4.8, labeling each state with the appropriate term.

4.2.4 For Example 4.12, write out the derivation of the transition system in Figure 2.6, labeling each state with the appropriate term.

4.2.5 For Example 4.17, write out the derivation of the transition system in Figure 2.9, labeling each state with the appropriate term.

4.2.6 For Example 4.18, argue that this specification has the same language as the specification in Example 4.17.

4.2.7 Write out the argument in Example 4.19 in more detail.

4.2.8 Construct the initial part of the automaton of the recursive specification in Example 4.20, and conclude it has the desired language.

4.2.9 Give a recursive specification over SA for the following languages ($n \geq 0, m \geq 0$):

(a) $L = \{a^n b^m \mid n \leq m + 3\}$;
(b) $L = \{a^n b^m \mid n \neq 2m\}$;
(c) $L = \{w \in \{a, b\}^* \mid \#_a(w) \neq \#_b(w)\}$;
(d) $L = \{w \in \{a, b\}^* \mid \#_a(w) = 2\#_b(w)\}$.

4.2.10 Give a recursive specification over SA for the following languages ($n \geq 0, m \geq 0, k \geq 0$):

(a) $L = \{a^n b^m c^k \mid n = m \text{ or } m \leq k\}$;
(b) $L = \{a^n b^m c^k \mid k = n + m\}$;
(c) $L = \{a^n b^m c^k \mid k \neq n + m\}$;
(d) $L = \{a^n b^m c^k \mid k \geq 3\}$;

4.2.11 Give a recursive specification over SA for the following languages:

(a) $L = \{a^n b^{2n} c^m \mid n > 0, m > 0\}$;
(b) $L = \{a^n b^m c^{2m+n} \mid n > 0, m > 0\}$;
(c) $L = \{a^n b^m c^i \mid 0 \leq n + m \leq i\}$;
(d) $L = \{a^m b^i a^n \mid i = m + n\}$;
(e) $L = \{wc^n \mid w \in \{a, b\}^*, n = \#_a(w) \vee n = \#_b(w)\}$.

4.2.12 Let $L = \{a^n b^n \mid n \geq 0\}$. Show that $L^2, \overline{L}$ and L^* are context-free.

4.2.13 Put $S = \mathbf{1} + a.S + S \cdot S$. Show this specification is unguarded. Using the operational rules, draw the transition system. Note this transition system is infinitely branching. Next, show this transition system is bisimilar to the transition system of $T = \mathbf{1} + a.T$.

4.2.14 Let $S = T \cdot (a.S + b.T)$ and $T = \mathbf{1} + a.T$. Show this specification is guarded. Using the operational rules, draw a transition system for this process.

4.2.15 Let $S = a.S + T$ and $T = b.T + S \cdot S$. Show this specification is unguarded. Draw a transition system for this process by means of the operational rules. This transition system is infinitely branching. Show the transition system is bisimilar to the transition system of $X = a.X+b.Y, Y = b.Y+a.X$. Thus, there is an equivalent guarded recursive specification.

4.2.16 Put $S = \mathbf{1} + a.S + T$ and $T = b.T + S \cdot S$. Show this specification is unguarded. Draw a transition system for this process by means of the operational rules.

4.2.17 Specialize the specifications of the stack for the case $\mathcal{D} = \{*\}$. Show the transition system of both processes are bisimilar (after reduction and relabeling) to the process of Example 4.17.

4.3 Parsing and ambiguity

In this section, we only look at context-free languages, so we will reason with language equivalence. Later, we will get back to processes defined by recursive specifications over SA and bisimulation equivalence.

As for regular languages, we can obtain each element $w \in \mathcal{L}(S)$ as a summand $w\mathbf{1}$ of the starting variable S by means of three principles:

1. Expansion: replacing a variable by its right-hand side;
2. Distribution: applying distributive laws;
3. Selection of a summand.

Besides the distributive laws, also other laws will be used freely without mention. In particular, by not writing brackets, we use implicitly the associativity of sequential composition and the compatibility of action prefix and sequential composition ($a.(x \cdot y) = (a.x) \cdot y$). Also the laws for $\mathbf{1}$ will be used often.

Example 4.23. Take the following (unguarded) recursive specification over SA:

$$\begin{aligned} S &= A \cdot B \\ A &= \mathbf{1} + a.a.A \\ B &= \mathbf{1} + B \cdot b.\mathbf{1}. \end{aligned}$$

Then we can see as follows that $aab \in \mathcal{L}(S)$:

$$S \approx A \cdot B \approx (\mathbf{1} + aaA) \cdot B \approx \mathbf{1} \cdot B + aaA \cdot B \gtrsim aaA \cdot B \approx aa(\mathbf{1} + aaA) \cdot B \approx$$

$$\approx aa\mathbf{1}\cdot B+aaaaA\cdot B \gtrsim aa\mathbf{1}\cdot B \approx aaB \approx aa(\mathbf{1}+B\cdot b\mathbf{1}) \approx aa\mathbf{1}+aaB\cdot b\mathbf{1} \gtrsim aaB\cdot b\mathbf{1} \approx$$

$$\approx aa(\mathbf{1} + B \cdot b\mathbf{1}) \cdot b\mathbf{1} \approx aa\mathbf{1} \cdot b\mathbf{1} + aaB \cdot b\mathbf{1} \cdot b\mathbf{1} \gtrsim aa\mathbf{1} \cdot b\mathbf{1} \approx aab\mathbf{1}.$$

In this derivation, each time the *left-most* variable was expanded. It is also possible to expand the *right-most* variable each time there is a choice, obtaining

$$S \approx A\cdot B \approx A\cdot(\mathbf{1}+B\cdot b\mathbf{1}) \approx A\cdot\mathbf{1}+A\cdot B\cdot b\mathbf{1} \gtrsim A\cdot B\cdot b\mathbf{1} \approx A\cdot(\mathbf{1}+B\cdot b\mathbf{1})\cdot b\mathbf{1} \approx$$

$$\approx A\cdot\mathbf{1}\cdot b\mathbf{1}+A\cdot B\cdot b\mathbf{1}\cdot b\mathbf{1} \gtrsim A\cdot b\mathbf{1} \approx (\mathbf{1}+aaA)\cdot b\mathbf{1} \approx \mathbf{1}\cdot b\mathbf{1}+aaA\cdot b\mathbf{1} \gtrsim aaA\cdot b\mathbf{1} \approx$$

$$\approx aa(\mathbf{1} + aaA) \cdot b\mathbf{1} \approx aa\mathbf{1} \cdot b\mathbf{1} + aaaaA \cdot b\mathbf{1} \gtrsim aab\mathbf{1}.$$

Whenever a string is in a language of a recursive specification, it has several derivations, in particular it has a left-most and a right-most derivation.

Presenting derivations like this, we usually just mention the sequential terms, and so the left-most derivation of $aab \in \mathcal{L}(S)$ is given as follows:

$$S \approx A \cdot B \gtrsim aaA \cdot B \gtrsim aa\mathbf{1} \cdot B \approx aaB \gtrsim aaB \cdot b\mathbf{1} \gtrsim aa\mathbf{1} \cdot b\mathbf{1} \approx aab\mathbf{1},$$

and the right-most derivation as follows:

$$S \approx A \cdot B \gtrsim A \cdot B \cdot b\mathbf{1} \gtrsim A \cdot b\mathbf{1} \gtrsim aaA \cdot b\mathbf{1} \gtrsim aa\mathbf{1} \cdot b\mathbf{1} \approx aab\mathbf{1}.$$

We will pay a lot of attention to derivations in this form. Each time the symbol $\gtrsim$ is used, a variable is replaced by one of its summands. Also in the first step, a variable is replaced by one of its summands. We call each such replacement *one step* in the derivation. Thus, in the present example, both the left-most and the right-most derivations take 5 steps.

Theorem 4.24. Let x be a term over SA with extra names $\mathcal{N}$. Then for all strings $w \in \mathcal{A}^*$:

$$w \in \mathcal{L}(x) \qquad \Longleftrightarrow \qquad x \gtrsim w\mathbf{1}.$$

Given a recursive specification E over SA with initial variable S, and a string $w \in \mathcal{A}^*$, we want to determine whether $w \in \mathcal{L}(S)$, i.e. we want to know whether there is a derivation $S \gtrsim w\mathbf{1}$. The procedure of determining whether or not there is such a derivation is called *parsing*.

The simplest way of parsing is to consider all possible left-most derivations and compare them with the string that is being parsed. This is called *exhaustive search parsing* or *brute force parsing*. We look at an example.

Example 4.25. Consider the (unguarded) recursive specification

$$S = \mathbf{1} + a.S \cdot b.\mathbf{1} + b.S \cdot a.\mathbf{1} + S \cdot S$$

and the string $w = aabb$. We will only look at sequential terms. As the first step, we can consider the four summands:

1. $S \gtrsim \mathbf{1}$,
2. $S \gtrsim aS \cdot b\mathbf{1}$,
3. $S \gtrsim bS \cdot a\mathbf{1}$,
4. $S \gtrsim S \cdot S$.

Of these four possibilities, the first and the third can never lead to $aabb$, so we can discard these derivations. The second and the fourth can be expanded, for the second we obtain as the second step:

1. $S \gtrsim a\mathbf{1} \cdot b\mathbf{1} \approx ab\mathbf{1}$,
2. $S \gtrsim a(aS \cdot b\mathbf{1}) \cdot b\mathbf{1} \approx aaS \cdot bb\mathbf{1}$,
3. $S \gtrsim a(bS \cdot a\mathbf{1}) \cdot b\mathbf{1} \approx abS \cdot ab\mathbf{1}$,

4. $S \gtrsim aS \cdot S \cdot b\mathbf{1}$.

The left-most derivations of the fourth possibility have the following as the second step:

1. $S \gtrsim \mathbf{1} \cdot S \approx S$,
2. $S \gtrsim (aS \cdot b\mathbf{1}) \cdot S \approx aS \cdot bS$,
3. $S \gtrsim (bS \cdot a\mathbf{1}) \cdot S \approx bS \cdot aS$,
4. $S \gtrsim S \cdot S \cdot S$.

Again, several possibilities can be discarded. In the third step, we find the string we are looking for as one of the eight possibilities:

$$S \gtrsim aS \cdot b\mathbf{1} \gtrsim aaS \cdot bb\mathbf{1} \gtrsim aa\mathbf{1} \cdot bb\mathbf{1} \approx aabb\mathbf{1},$$

and thus we see $aabb \in \mathcal{L}(S)$.

We see that this is a cumbersome method of determining whether a certain string is in a language given by a recursive specification, as there can be very many derivations. When a string is in the language, eventually a derivation will be found, but if it is not, the procedure may go on indefinitely, never producing a result. This is the reason we will be looking at recursive specifications in a particular form, so that we are sure that parsing will always produce a result in a number of steps.

Suppose that we have given a recursive specification where every right-hand side of every variable does not contain a summand $\mathbf{1}$ and does not contain a summand which is a single variable or contains a $\mathbf{0}$. Then, every summand will contain at least one element of $\mathcal{A}$ or at least twice an element of $\mathcal{N}$. As a result, every sequential term produced by the procedure above will make progress towards the string to be parsed: every time a summand is chosen with one alphabet element one more element of the resulting string is fixed, and every time a summand is chosen with two variables, the length of the resulting string will need to increase.

Example 4.26. Consider the (unguarded) recursive specification

$$S = a.b.\mathbf{1} + b.a.\mathbf{1} + a.S \cdot b.\mathbf{1} + b.S \cdot a.\mathbf{1} + S \cdot S.$$

It has the same language as the previous example, except that it does not accept the empty string. Every summand of the right-hand side has two elements of $\mathcal{A}$ or twice an element of $\mathcal{N}$. Then, given any string w, we know at least after $|w|$ rounds in the procedure above whether or not it is in the language.

In general, whenever a recursive specification has no $\mathbf{1}$ summands, no $\mathbf{0}$-containing summands and no single variable summands, then given a string in $\mathcal{A}^*$ we know at least after $2|w|$ rounds whether or not this string is in the language. As an example, for the recursive specification $S = a\mathbf{1} + b\mathbf{1} + S \cdot S$, every $w \in \{a, b\}^*$ is in $\mathcal{L}(S)$, and the derivation $S \gtrsim w\mathbf{1}$ has exactly length $2|w| - 1$.

We will see later that every recursive specification can be rewritten using language equivalence such that summands containing $\mathbf{0}$ of the form $\mathbf{1}$ or a single variable do not occur (disregarding the empty string). Then parsing by exhaustive search can always be done. Still, the method can be very inefficient, and many methods have been developed to do parsing more efficiently in many cases.

We mention one of these cases explicitly.

Definition 4.27. Let E be a recursive specification over SA in sequential form with added names $\mathcal{N}$. If for every $P \in \mathcal{N}$, the right-hand side of P only contains summands of the form $a.X$ (so is guarded), where X is a sequential composition of names in $\mathcal{N}$, and every summand of P starts with a different $a \in \mathcal{A}$, then we call E *simple*.

Example 4.28. The (guarded) specification $S = c.\mathbf{1} + a.S + b.S \cdot S$ is simple, as $\mathbf{1}$ is considered as the sequential composition of zero variables. The (guarded) specification $S = c.\mathbf{1} + a.S + b.S \cdot S + c.S \cdot S$ is not simple, as there are two summands starting with c.

Given a simple recursive specification, a string w can be parsed in $|w|$ steps: at every step we know which summand to take, and at every step exactly one extra element of $\mathcal{A}$ is determined. Unfortunately, it is not the case that every recursive specification over SA is language equivalent to a simple one.

Whenever a string is in the language given by a recursive specification, there is a left-most derivation for it. Sometimes, there is more than one left-most derivation. This is called ambiguity.

Definition 4.29. Let E be recursive specification over SA, then E is called *ambiguous* if some string w in the language of E has more than one left-most derivation.

Example 4.30. Consider the specification given by $S = \mathbf{1} + a.S \cdot b.\mathbf{1} + S \cdot S$. The string $aabb$ has two different left-most derivations:

$$S \gtrsim aS \cdot b\mathbf{1} \gtrsim aaS \cdot b\mathbf{1} \cdot b\mathbf{1} \approx aaS \cdot bb\mathbf{1} \gtrsim aa\mathbf{1} \cdot bb\mathbf{1} \approx aabb\mathbf{1},$$

and

$$S \gtrsim S \cdot S \gtrsim \mathbf{1} \cdot S \gtrsim aS \cdot b\mathbf{1} \gtrsim aaS \cdot bb\mathbf{1} \gtrsim aa\mathbf{1} \cdot bb\mathbf{1} \approx aabb\mathbf{1}.$$

Here, ambiguity is quite harmless but there are situations where different parsing leads to different results. In arithmetic, an expression $a * b + c$ can be parsed as $(a * b) + c$ or as $a * (b + c)$. This can be expressed in terms of a recursive specification as follows, writing m for multiplication and p for addition. We have $\mathcal{N} = \{E, I\}$ (Expression, Identifier) and $\mathcal{A} = \{a, b, c, m, p, (,)\}$. Note that brackets are used as elements of $\mathcal{A}$.

$$\begin{aligned} E &= I + E \cdot m.E + E \cdot p.E + (.E\cdot).\mathbf{1} \\ I &= a.\mathbf{1} + b.\mathbf{1} + c.\mathbf{1} \end{aligned}$$

We have the following derivations:

$$E \gtrsim E \cdot mE \gtrsim a\mathbf{1} \cdot mE \cdot pE \gtrsim amb\mathbf{1} \cdot pc\mathbf{1} \approx ambpc\mathbf{1},$$

$$E \gtrsim E \cdot pE \gtrsim E \cdot mE \cdot pc\mathbf{1} \gtrsim a\mathbf{1} \cdot mb\mathbf{1} \cdot pc\mathbf{1} \approx ambpc\mathbf{1}.$$

The first one begins with a multiplication, the second one begins with an addition.

The given recursive specification can be disambiguated by introducing extra variables T, F (Term, Factor) as follows:

$$\begin{aligned} E &= T + E \cdot p.T \\ T &= F + T \cdot m.F \\ F &= I + (.E\cdot).\mathbf{1} \\ I &= a.\mathbf{1} + b.\mathbf{1} + c.\mathbf{1} \end{aligned}$$

Now string *ambpc* allows only one derivation:

$$E \gtrsim E{\cdot}pT \gtrsim T{\cdot}pF \gtrsim T{\cdot}mF{\cdot}pI \gtrsim F{\cdot}mI{\cdot}pc\mathbf{1} \gtrsim I{\cdot}mb\mathbf{1}{\cdot}pc\mathbf{1} \gtrsim a\mathbf{1}{\cdot}mbpc\mathbf{1} \approx ambpc\mathbf{1}.$$

However, such a disambiguation is not always possible. We can give an example of a specification that has ambiguity that cannot be removed, but we cannot prove at this point that this is the case.

Example 4.31. Consider the guarded recursive specification

$$\begin{aligned} S &= T + U \\ T &= V \cdot C \\ U &= A \cdot W \\ V &= \mathbf{1} + a.V \cdot b.\mathbf{1} \\ W &= \mathbf{1} + b.W \cdot c.\mathbf{1} \\ A &= \mathbf{1} + a.A \\ C &= \mathbf{1} + c.C. \end{aligned}$$

We see $\mathcal{L}(T) = \{a^n b^n c^m \mid n, m \geq 0\}$ and $\mathcal{L}(U) = \{a^m b^n c^n \mid n, m \geq 0\}$, so $\mathcal{L}(S)$ is the union of these two. Now any string $a^n b^n c^n \in \mathcal{L}(S)$ has two different derivations, one starting with $S \gtrsim T$, the other starting with $S \gtrsim U$. We claim (without proof) that there is no disambiguous recursive specification over SA with language $\mathcal{L}(S)$. This has to do with the fact that the language $\{a^n b^n c^n \mid n \geq 0\}$ is not context-free, a fact that we will prove later.

In order to define the syntax of programming languages, often recursive specifications are used, usually in the form of *Backus-Naur Form* or BNF. In this presentation, variables are identifiers enclosed by angle brackets, ::= is used instead of =, | instead of +, the $\cdot$ symbol is left out and just a is written for $a.\mathbf{1}$. Thus, the arithmetic specification of Example 4.30 could be written as follows:

$$\begin{aligned} \langle expression \rangle &::= \langle term \rangle \mid \langle expression \rangle p \langle term \rangle \\ \langle term \rangle &::= \langle factor \rangle \mid \langle term \rangle m \langle factor \rangle \\ \langle factor \rangle &::= \langle identifier \rangle \mid (\langle expression \rangle) \\ \langle identifier \rangle &::= a \mid b \mid c \end{aligned}$$

Exercises

4.3.1 Give a derivation of $aabbbb \in \mathcal{L}(S)$, if S is given by

$$\begin{aligned} S &= \mathbf{1} + A \cdot B \\ A &= a.B \\ B &= S \cdot b.\mathbf{1}. \end{aligned}$$

Can you describe $\mathcal{L}(S)$?

4.3.2 Show that $aabbabba \notin \mathcal{L}(S)$, if S is given by

$$\begin{aligned} S &= a.a.B \\ A &= \mathbf{1} + b.B \cdot b.\mathbf{1} \\ B &= A \cdot a.\mathbf{1}. \end{aligned}$$

4.3.3 Find a simple recursive specification for $\mathcal{L}(a.a.(a.\mathbf{1})^* \cdot b.\mathbf{1} + b.\mathbf{1})$.

4.3.4 Find a simple recursive specification for $L = \{a^n b^n \mid n > 0\}$.

4.3.5 Show that every simple recursive specification is unambiguous.

4.3.6 Show that the following recursive specification is ambiguous. Then construct an unambiguous recursive specification that has the same language.

$$\begin{aligned} S &= A \cdot B + a.a.B \\ A &= a.\mathbf{1} + A \cdot a.\mathbf{1} \\ B &= b.\mathbf{1} \end{aligned}$$

4.3.7 Show that every regular language has an unambiguous specification.

4.3.8 Show that the specification $S = \mathbf{1} + a.S \cdot b.S + b.S \cdot a.S$ is ambiguous.

4.4 Simplification of specifications and normal forms

Sometimes, it has advantages of having a recursive specification in a particular form. In the chapter on finite automata, a specification in linear form had a more direct interpretation as an automaton. In this chapter, it was advantageous to present recursive specifications in sequential form. Considering parsing, we could enforce progress by assuming that every summand of every right-hand side was not $\mathbf{1}, \mathbf{0}$ or a single variable.

We start out with a simple observation that is valid in bisimilarity. Every sequential term can be written as a sequential composition of variables, by adding extra variables if necessary. Thus, if we have a term $a.B \cdot c.D$, add variables $A = a.\mathbf{1}$ and $C = c.\mathbf{1}$ and we can write $A \cdot B \cdot C \cdot D$.

Next, we describe a simple procedure valid in language equivalence, but not in bisimilarity. This procedure is called *removing zeroes*. By using the laws of language equivalence, any sequential term containing $\mathbf{0}$ can be replaced by $\mathbf{0}$

(in bisimilarity, a term like $a.\mathbf{0}$ cannot be simplified). Next, as long as there are other summands on the right-hand side of an equation, a $\mathbf{0}$ summand can be left out. If $P = \mathbf{0}$ is an equation, P can be replaced by $\mathbf{0}$ throughout, and the equation can be left out. The procedure can then be repeated, until all $\mathbf{0}$ are removed (the specification might become empty, have no equations left). To give an example, removing zeroes in

$$\begin{aligned} S &= A \cdot B + a.a.A \\ A &= A \cdot \mathbf{0} \cdot A + A \cdot a.\mathbf{1} \\ B &= b.\mathbf{0} \end{aligned}$$

leads to

$$\begin{aligned} S &\approx a.a.A \\ A &\approx A \cdot a.\mathbf{1}. \end{aligned}$$

In the sequel, we usually assume a specification we start out from contains no zeroes.

Next, we consider a simple procedure valid in bisimilarity. This procedure is *removing unreachable variables*. Variables that cannot be reached from the initial variable can be left out, together with their equations. Consider the following example.

Example 4.32. Consider the recursive specification

$$\begin{aligned} S &= \mathbf{1} \cdot A \cdot b.\mathbf{1} \\ A &= A \cdot S + B \cdot \mathbf{1} \\ B &= B + S \\ C &= \mathbf{1} + c.C. \end{aligned}$$

From S, A can be reached, as it occurs in the right-hand side of the equation of S. From A, A, S, B can be reached, and from B, B, S can be reached. We see C can never be reached, and its equation can be left out.

Theorem 4.33. Let E be a recursive specification over SA with initial variable S. Then S is bisimilar to the initial variable of the specification obtained by leaving out all equations of unreachable variables.

Proof. The set of reachable variables can be determined as follows:

1. A variable that occurs in a sequential summand of the initial variable is reachable;

2. A variable that occurs in a sequential summand of a variable that can be reached from the initial variable, is also reachable.

In this way, the set of reachable variables can be determined inductively.

Finding the transition system by means of the operational rules, every reachable state will be denoted by a term that only contains reachable variables. Never, a term containing an unreachable variable can occur. □

Another procedure valid in bisimilarity is *removing single variable summands*. If the right-hand side of the equation of P has a summand Q ($P, Q \in \mathcal{N}$), so $P \gtrsim Q$, we say P has a single variable summand Q. This was already dealt with in Theorem 2.39. We just give an example.

Example 4.34. Consider the following recursive specification:

$$\begin{aligned} S &= A \cdot a.\mathbf{1} + B \\ B &= A + b.b.\mathbf{1} \\ A &= a.\mathbf{1} + b.c.\mathbf{1} + B. \end{aligned}$$

Now, expand all single variable summands until no new summands are obtained, and then leave them out.

$$\begin{aligned} S &\leftrightarrows A \cdot a.\mathbf{1} + A + b.b.\mathbf{1} \leftrightarrows A \cdot a.\mathbf{1} + a.\mathbf{1} + b.c.\mathbf{1} + B + b.b.\mathbf{1} \leftrightarrows \\ &\leftrightarrows A \cdot a.\mathbf{1} + a.\mathbf{1} + b.c.\mathbf{1} + b.b.\mathbf{1} \\ B &\leftrightarrows a.\mathbf{1} + b.c.\mathbf{1} + B + b.b.\mathbf{1} \leftrightarrows a.\mathbf{1} + b.c.\mathbf{1} + b.b.\mathbf{1} \\ A &\leftrightarrows a.\mathbf{1} + b.c.\mathbf{1} + A + b.b.\mathbf{1} \leftrightarrows a.\mathbf{1} + b.c.\mathbf{1} + b.b.\mathbf{1}. \end{aligned}$$

Next, we consider the *Greibach normal form*.

Definition 4.35. Let a recursive specification E over SA with added names $\mathcal{N}$ be given. We say E is in *Greibach normal form* if each right-hand side of each equation has one of the following forms:

1. The right-hand side is $\mathbf{0}$ or $\mathbf{1}$;
2. The right-hand side consists of a number of summands, each of which is either $\mathbf{1}$ or of the form $a.X$, with $a \in \mathcal{A}$ and X a sequential composition of variables, $X \in \mathcal{N}^*$.

Theorem 4.36. Let E be a *guarded* recursive specification over SA. Then E is bisimulation equivalent to a specification in Greibach normal form.

Proof. By removing single variable summands, we can assume every right-hand side is a sum of sequential terms, each of which is a constant or contains at least two elements. By expansion and distribution, we can write each sequential term so that it starts with an action (here, we use guardedness). The final step is adding extra variables, in case a prefixing occurs within a sequential term, so if we have $a.A \cdot b.C$, add $B = b.\mathbf{1}$ and write $a.A \cdot B \cdot C$. □

In case we start out from an *unguarded* recursive specification, this procedure fails, as we cannot make progress when a variable P has a sequential summand that starts with P. This is called *head recursion*. We cannot continue under bisimulation, but we can continue under language equivalence. We will not treat this procedure in detail, but just provide a couple of examples.

We look at a simple example: suppose we have an equation $A = A \cdot B + a.\mathbf{1}$. Then, if there is a derivation $A \gtrsim A \cdot B \gtrsim w\mathbf{1}$ for some $w \in \mathcal{L}(A)$, then at some point in this derivation A must be expanded with $a.\mathbf{1}$. Then we may as well do this expansion first, and do the rest of the derivation following. This means we can replace the given equation by:

$$\begin{aligned} A &\approx a.C \\ C &\approx \mathbf{1} + B \cdot C, \end{aligned}$$

where C is some new variable. We see C can generate any number of B's that was generated by expanding A using the first summand. We call this procedure *removal of head recursion*.

To give another example, if $A = A \cdot B + a.D$, then we replace by

$$\begin{aligned} A &\approx a.D \cdot C \\ C &\approx \mathbf{1} + B \cdot C. \end{aligned}$$

A final example: if $A = A \cdot B + A \cdot B' + D \cdot E + a.\mathbf{1} + b.\mathbf{1}$, replace by

$$\begin{aligned} A &\approx D \cdot E \cdot C + a.C + b.C \\ C &\approx \mathbf{1} + B \cdot C + B' \cdot C. \end{aligned}$$

Theorem 4.37. Let E be a recursive specification over SA. Then there is a guarded language equivalent recursive specification F.

Proof. The examples provided can be turned into a general procedure. We leave out the details. □

In the literature, Greibach normal form is usually further restricted and does not allow $\mathbf{1}$ summands also. This can be achieved in addition with the procedure of removing $\mathbf{1}$ summands. This procedure is only valid in language equivalence.

The advantage of the Greibach normal form without $\mathbf{1}$ summands is that it is straightforward to generate the corresponding transition system. Every state in the transition system is a sequence of variables, and the outgoing edges of any state correspond directly to the summands of the leading variable.

Thus, every state of a process in Greibach normal form without $\mathbf{1}$ summands can be seen as a sequence of variables. If the first variable of such a sequence does not have a $\mathbf{1}$ summand, then the following variables cannot contribute an outgoing step to the state, and the number of outgoing steps is the number of summands of the leading variable. Thus, if such a process is given by a guarded recursive specification over variables $\mathcal{N}$, and for all states $X \in \mathcal{N}^*$ with more than one variable, the leading variable does not have a $\mathbf{1}$ summand, then for all states the number of outgoing steps is the number of summands of the leading variable. In such a case, the process has bounded branching (as a bound, take the maximum of the number of summands of all variables).

Also conversely, if a process defined by a guarded recursive specification over SA has bounded branching, then it has a recursive specification over SA, where every state either is a single variable or the leading variable does not have a $\mathbf{1}$ summand. We will not prove this, but just provide an example.

Example 4.38. For the stack process of Example 4.10, the initial state has $|\mathcal{D}|$ outgoing steps (labeled $i?d$ for every $d \in \mathcal{D}$), and all other states have $|\mathcal{D}| + 1$ outgoing steps (besides all inputs the output of the top of the stack). Thus, the stack has bounded branching. Nevertheless, there is a state $S \cdot !d.S$ where S has a $\mathbf{1}$ summand.

We can remedy this by defining new variables T_d (for every $d \in \mathcal{D}$) by $T_d = S \cdot !d.\mathbf{1}$. Then, we get a new specification over the extended set of variables as follows:

$$\begin{aligned} S &= \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.T_d \cdot S \\ T_d &= o!d.\mathbf{1} + \sum_{e \in \mathcal{D}} i?e.T_e \cdot T_d. \end{aligned}$$

The state where the stack contents is $d_1 d_2 \ldots d_n$ is now given by the sequence of variables $T_{d_1} \cdot \ldots \cdot T_{d_n} \cdot S$, and the T_d do not have $\mathbf{1}$ summands.

On occasion, we will need to restrict the Greibach normal form further, in a different way.

Definition 4.39 (—Restricted Greibach normal form)**.** We say a recursive specification in Greibach normal form is *restricted* if for every summand $a.X$, the sequence $X \in \mathcal{N}^*$ contains at most two variables.

It is straightforward to reduce a recursive specification in Greibach normal form to one in Restricted Greibach normal form: we add a new variable for every sequence of two variables occurring in some summand and do substitutions from left to right. In this way, the length of all sequences is reduced by at least one.

We continue with procedures preserving only language equivalence, not bisimilarity. First of all, we remark that often, we will assume that the empty string is not in the language considered. This is not an important restriction: if E is a recursive specification with initial variable S, then on the one hand, adding a new equation $S' = \mathbf{1} + S$ to E will result in $\mathcal{L}(S') = \mathcal{L}(S) \cup \{\varepsilon\}$; on the other hand, the procedure *removing* $\mathbf{1}$*-summands* to be discussed further on will result in the language $\mathcal{L}(S) - \{\varepsilon\}$.

We first discuss the procedure *removing nonproductive variables*.

Example 4.40. Consider the recursive specification

$$\begin{aligned} S &= \mathbf{1} + a.S \cdot b.\mathbf{1} + A \\ A &= a.A \\ B &= \mathbf{1}. \end{aligned}$$

The variable A is nonproductive, as from A, a final state can never be reached. Remark that B is unreachable.

Definition 4.41. Let E be a recursive specification over SA with variables $\mathcal{N}$ and initial variable $S \in \mathcal{N}$. A variable $P \in \mathcal{N}$ is called *productive* iff there is a string $w \in \mathcal{L}(S)$ such that $P \gtrsim w\mathbf{1}$.

This means a productive variable has a terminating derivation. A variable that is not productive is called *nonproductive*. A sequential term containing a nonproductive variable (or a $\mathbf{0}$) can never have a terminating derivation.

Example 4.42. We show how to remove nonproductive variables from the following specification:

$$\begin{aligned} S &= a.S + A + C \\ A &= a.\mathbf{1} \\ B &= a.a.\mathbf{1} \\ C &= a.C \cdot b.\mathbf{1}. \end{aligned}$$

As before, we assume that the specification is in sequential form. We identify which variables have a terminating derivation. From the right-hand sides of A and B we see immediately that they are productive. Also, S can lead to A, so also S is productive. On the other hand, C only leads to C, and each derivation

starting from C necessarily contains C. This identifies C as nonproductive. Removing C and all summands containing C leads to the specification

$$\begin{aligned} S &\approx a.S + A \\ A &\approx a.\mathbf{1} \\ B &\approx a.a.\mathbf{1}. \end{aligned}$$

Next, we can also remove unreachable variables. The right-hand side of S tells us we can reach S and A. From A, no further variables can be reached. Thus, we see B can never be reached from S, and we can remove B, resulting in the specification

$$\begin{aligned} S &\approx a.S + A \\ A &\approx a.\mathbf{1}. \end{aligned}$$

We formulate this result in the form of a theorem.

Theorem 4.43. Let E be a recursive specification over SA in sequential form. There is a language equivalent recursive specification F over SA that does not contain any nonproductive variables.

Proof. Use the procedure outlined above to construct F from E, leaving out the variables that are nonproductive and all sequential terms containing these variables.

To be more precise, we determine the set of productive variables as follows:

1. A variable with a sequential summand of the form $w\mathbf{1}$ is productive;
2. A variable with a sequential summand in which all occurring variables are productive, is also productive.

In this way, the set of productive variables can be determined inductively.

Now take any string w in $\mathcal{L}(S)$, where S is the initial variable of E. Then there is a derivation $S \gtrsim w\mathbf{1}$. Now every variable occurring in this derivation is productive, so this derivation is entirely inside F. Conversely, every derivation in F is trivially inside E, so E and F have the same language. □

Next, we look at the procedure *removing* $\mathbf{1}$*-summands*. This procedure only works when ε is not in the language of the recursive specification under consideration.

Example 4.44. Consider the specification

$$\begin{aligned} S &= a.T \cdot b.\mathbf{1} \\ T &= \mathbf{1} + a.T \cdot b.\mathbf{1}. \end{aligned}$$

Notice $\varepsilon \notin \mathcal{L}(S)$, in fact $\mathcal{L}(S) = \{a^n b^n \mid n \geq 1\}$. We can remove the $\mathbf{1}$-summand of T after adding, for each summand containing T on the right-hand side, a new summand with T replaced by $\mathbf{1}$:

$$\begin{aligned} S &\approx a.T \cdot b.\mathbf{1} + a.b.\mathbf{1} \\ T &\approx a.T \cdot b.\mathbf{1} + a.b.\mathbf{1}. \end{aligned}$$

The resulting specification has the same language.

Definition 4.45. A variable P in a recursive specification E is called *nullable* iff $P \gtrsim \mathbf{1}$.

Notice that variable S is nullable iff $\varepsilon \in \mathcal{L}(S)$.

The set of nullable variables can be determined inductively:

1. A variable with a $\mathbf{1}$ summand is nullable;
2. A variable with a summand that is a sequential composition of nullable variables, is also nullable.

Example 4.46. Consider the specification

$$\begin{aligned} S &= A \cdot B \cdot a.C \\ A &= B \cdot C \\ B &= \mathbf{1} + b.\mathbf{1} \\ C &= \mathbf{1} + D \\ D &= d.\mathbf{1}. \end{aligned}$$

First, we determine the set of nullable variables. As B and C have a $\mathbf{1}$-summand, they are nullable. Next, $A \gtrsim B \cdot C \gtrsim \mathbf{1} \cdot \mathbf{1} \approx \mathbf{1}$, so also A is nullable. As any sequential term derived from S contains an a, S is not nullable (and so $\varepsilon \notin \mathcal{L}(S)$). Obviously, D is not nullable, it has only one derivation.

Next, for each sequential term on the right-hand side, we add a new summand, replacing each combination of nullable variables in this term by $\mathbf{1}$:

$$\begin{aligned} S &\approx A \cdot B \cdot a.C + \mathbf{1} \cdot B \cdot a.C + A \cdot \mathbf{1} \cdot a.C + A \cdot B \cdot a.\mathbf{1} + \\ &\quad + A \cdot \mathbf{1} \cdot a.\mathbf{1} + \mathbf{1} \cdot B \cdot a.\mathbf{1} + \mathbf{1} \cdot \mathbf{1} \cdot a.C + \mathbf{1} \cdot \mathbf{1} \cdot a.\mathbf{1} \approx \\ &\approx A \cdot B \cdot a.C + B \cdot a.C + A \cdot a.C + A \cdot B \cdot a.\mathbf{1} + A \cdot a.\mathbf{1} + B \cdot a.\mathbf{1} + a.C + a.\mathbf{1} \\ A &\approx B \cdot C + \mathbf{1} \cdot C + B \cdot \mathbf{1} + \mathbf{1} \cdot \mathbf{1} \approx B \cdot C + B + C + \mathbf{1} \\ B &\approx \mathbf{1} + b.\mathbf{1} \\ C &\approx \mathbf{1} + D \\ D &\approx d.\mathbf{1}. \end{aligned}$$

Next, remove all $\mathbf{1}$-summands:

$$\begin{aligned} S &\approx A \cdot B \cdot a.C + B \cdot a.C + A \cdot a.C + A \cdot B \cdot a.\mathbf{1} + A \cdot a.\mathbf{1} + B \cdot a.\mathbf{1} + a.C + a.\mathbf{1} \\ A &\approx B \cdot C + B + C \\ B &\approx b.\mathbf{1} \\ C &\approx D \\ D &\approx d.\mathbf{1}. \end{aligned}$$

The resulting specification does not have any $\mathbf{1}$-summands, and has the same language as the starting specification.

Theorem 4.47. Let E be a recursive specification over SA with initial variable S and $\varepsilon \notin \mathcal{L}(S)$. Then there is a language equivalent recursive specification F without $\mathbf{1}$-summands.

Proof. Obtain F from E by following the procedure given above. Consider any derivation $S \gtrsim w\mathbf{1}$, where S is the initial variable of E. Now whenever in this derivation a step is taken by replacing a variable, say A, by $\mathbf{1}$, go back to the step where this variable is introduced in the derivation, and replace A by $\mathbf{1}$ in this step. If necessary, this is repeated. The result is a derivation in F, showing that F and E have the same language. □

Above, we already saw the procedure *removing single variable summands* and the procedure *removing unreachable variables*. Since these procedure preserve bisimilarity, they certainly preserve language equivalence. Putting everything together, we obtain the following theorem.

Theorem 4.48. Let L be a context-free language with $\varepsilon \notin L$. Then there is a recursive specification over SA with initial variable S that has $\mathcal{L}(S) = L$ without any zeroes, any nonproductive or unreachable variables, $\mathbf{1}$-summands or single variable summands.

Proof. As L is context-free, there is a recursive specification E with initial variable S such that $\mathcal{L}(S) = L$.

The procedures just discussed need to be applied in the correct order. First, remove all zeroes and all $\mathbf{1}$-summands of E, next remove all single variable summands, then remove all nonproductive variables and finally remove all unreachable variables. □

We see that we can modify a recursive specification such that all its right-hand summands are in a particular form. Much more can be done in this direction. We finish this section by looking at the so-called *Chomsky normal form*.

Definition 4.49. A recursive specification over SA is said to be *in Chomsky normal form* if all right-hand summands are either a sequential composition of two variables, $P \cdot Q$, or a single letter of the alphabet, $a.\mathbf{1}$.

Theorem 4.50. Let E be a recursive specification over SA with initial variable S and $\varepsilon \notin \mathcal{L}(S)$. Then there is a language equivalent recursive specification F in Chomsky normal form.

Proof. We can assume that E has no zeroes, no $\mathbf{1}$-summands and no single variable summands. In the first step, transform E so that every summand is either of the form $a.\mathbf{1}$ or is a sequential composition of two or more variables. We do this by introducing new variables, so e.g. if there is a summand $a.a.B \cdot c.\mathbf{1}$, introduce variables $A = a.\mathbf{1}$ and $C = c.\mathbf{1}$ and write the summand as $A \cdot A \cdot B \cdot C$. It is obvious that the transformed specification has the same language.

In the second step, again by introducing new variables, we reduce the length of the sequential compositions of variables. If e.g. there is a summand $A \cdot B \cdot C \cdot D$, introduce new variables $E = B \cdot F$, $F = C \cdot D$ and replace the summand by $A \cdot E$. Again, the transformed specification has the same language. Moreover, it is in Chomsky normal form. □

We give an example.

Example 4.51. Consider the following recursive specification:

$$\begin{aligned} S &= A \cdot B \cdot a.\mathbf{1} \\ A &= a.a.b.\mathbf{1} \\ B &= A \cdot c.\mathbf{1}. \end{aligned}$$

Note that ε is not in the language, as every derivation of S contains an a. Moreover, there are no $\mathbf{1}$-summands and no single variable summands. In the first step, we need new variables for a, b, c, call these A', B', C:

$$\begin{aligned} S &\leftrightarrows A \cdot B \cdot A' \\ A &\leftrightarrows A' \cdot A' \cdot B' \\ B &\leftrightarrows A \cdot C \\ A' &\leftrightarrows a.\mathbf{1} \\ B' &\leftrightarrows b.\mathbf{1} \\ C &\leftrightarrows c.\mathbf{1}. \end{aligned}$$

In the second step, we need new variables D, E:

$$\begin{aligned} S &\leftrightarrows A \cdot D \\ A &\leftrightarrows A' \cdot E \\ B &\leftrightarrows A \cdot C \\ A' &\leftrightarrows a.\mathbf{1} \\ B' &\leftrightarrows b.\mathbf{1} \\ C &\leftrightarrows c.\mathbf{1} \\ D &\leftrightarrows B \cdot A' \\ E &\leftrightarrows A' \cdot B'. \end{aligned}$$

We now have a specification in Chomsky normal form.

Exercises

4.4.1 Given is the specification

$$\begin{aligned} S &= a.b.A \cdot B + b.a.\mathbf{1} \\ A &= a.a.a.\mathbf{1} \\ B &= a.A + b.b.\mathbf{1}. \end{aligned}$$

Now show

$$\begin{aligned} S &\approx a.b.A \cdot a.A + a.b.A \cdot b.b.\mathbf{1} + b.a.\mathbf{1} \\ A &\approx a.a.a.\mathbf{1}. \end{aligned}$$

4.4.2 Remove all nonproductive and unreachable variables from the following specification:

$$\begin{aligned} S &= a.S + A \cdot B \\ A &= b.A \\ B &= A \cdot A. \end{aligned}$$

What is the language generated?

4.4.3 Remove nonproductive and unreachable variables from

$$\begin{aligned} S &= a.\mathbf{1} + a.A + B + C \\ A &= \mathbf{1} + a.B \\ B &= A \cdot a.\mathbf{1} \\ C &= c.C \cdot D \\ D &= d.d.d.\mathbf{1}. \end{aligned}$$

4.4.4 Remove $\mathbf{1}$ summands from

$$\begin{aligned} S &= A \cdot a.B + a.a.B \\ A &= \mathbf{1} \\ B &= \mathbf{1} + b.b.A. \end{aligned}$$

4.4.5 Remove all single variable summands, nonproductive and unreachable variables and $\mathbf{1}$ summands from

$$\begin{aligned} S &= a.A + a.B \cdot B \\ A &= \mathbf{1} + a.a.A \\ B &= b.B + b.b.C \\ C &= B. \end{aligned}$$

What language does this specification generate?

4.4.6 Give an example of a specification that has no single variable summands, but where removal of $\mathbf{1}$ summands leads to a single variable summand.

4.4.7 Suppose we have a specification without $\mathbf{1}$ summands. Show that removal of single variable summands cannot lead to a new $\mathbf{1}$ summand.

4.4.8 Suppose we have a specification without $\mathbf{1}$ summands or single variable summands. Show that removal of nonproductive and unreachable variables does not lead to $\mathbf{1}$ summands or single variable summands.

4.4.9 In Theorem 4.48, we first removed nonproductive variables, and then unreachable variables. If we do this in the reverse order, so first removing unreachable variables, and then removing nonproductive variables, can the resulting specification contain an unreachable variable? If not, prove this. If so, give an example where this occurs.

4.4.10 Transform the following recursive specifications into Chomsky normal form:

$$\begin{aligned} S &= a.b.A \cdot B \\ A &= b.A \cdot B + \mathbf{1} \\ B &= \mathbf{1} + A + B \cdot A \cdot a.\mathbf{1}, \end{aligned}$$

and

$$\begin{aligned} S &= A \cdot B + a.B \\ A &= \mathbf{1} + a.a.b.\mathbf{1} \\ B &= b.b.A. \end{aligned}$$

4.4.11 Transform the following recursive specifications into Greibach normal form:

$$\begin{aligned} S &= \mathbf{1} + S \cdot A + B \\ A &= a.A + b.c.\mathbf{1} \\ B &= B \cdot B \end{aligned}$$

and

$$\begin{aligned} S &= a.\mathbf{1} + A \cdot B \\ A &= b.\mathbf{1} + S \cdot A \\ B &= B \cdot S. \end{aligned}$$

4.4.12 Say $S = \mathbf{1} + a.S \cdot S$. This specification is guarded. Using the operational rules, draw a transition system for this process. Show the transition system generated has unbounded branching. Nevertheless, it is branching bisimilar to the system of $U \leftrightarrows \mathbf{1} + a.U$.

4.5 Push-down and context-free

Now we prove that the set of context-free languages is exactly the same as the set of push-down languages.

Theorem 4.52. Let L be a context-free language. Then there is a push-down automaton M with $\mathcal{L}(M) = L$.

Proof. Let L be a context-free language over alphabet $\mathcal{A}$. This means there is a recursive specification over SA with initial variable S and $\mathcal{L}(S) = L$. Transform this recursive specification into Greibach normal form, using variables $\mathcal{N}$ (this transformation is not really necessary, the following procedure can also be defined directly but is then less clear). Now define a push-down automaton $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ as follows:

1. $\mathcal{S} = \{\uparrow, \downarrow\}$;
2. $\mathcal{D} = \mathcal{N}$;
3. $\uparrow \stackrel{\varepsilon,\tau,S}{\longrightarrow} \downarrow$;
4. For each summand $a.X$ in the right-hand side of variable P, add a step $s \stackrel{P,a,X}{\longrightarrow} s$ (for $a \in \mathcal{A}, X \in \mathcal{N}^*$);
5. For each summand $\mathbf{1}$ in the right-hand side of variable P, add a step $s \stackrel{P,\tau,\varepsilon}{\longrightarrow} s$;

6. $\downarrow$ is a final state.

Now each execution of this push-down automaton can be matched with a summand derivation of S. Notice we can only say this in language equivalence, since we cannot be sure that all τ-steps introduced for $\mathbf{1}$-summands are inert in the transition system. Thus, the languages coincide. □

Theorem 4.53. Let M be a push-down automaton. Then there is a recursive specification over SA with initial variable S such that $\mathcal{L}(S) = \mathcal{L}(M)$.

Proof. This proof is quite complicated. In a given push-down automaton, the possibilities given for a certain top element of the stack will depend on the state the automaton is in, and therefore it is not enough to have a variable for each stack element. Instead, we need a variable for each stack element, combined with a state started from and the state where this stack element is removed from the stack.

First of all, we can restrict a given push-down automaton M in the following way:

1. M has exactly one final state $\downarrow$, and this state is only entered when the stack content is ε;

2. M has only push and pop transitions (see Theorem 4.10).

Given these restrictions, we will construct a recursive specification that has variables $V_{s\varepsilon}$ and V_{sdt}, for $s, t \in \mathcal{S}$ and $d \in \mathcal{D}$, such that $V_{s\varepsilon} \gtrsim w\mathbf{1}$ resp. $V_{sdt} \gtrsim w\mathbf{1}$ will hold exactly when $(s, \varepsilon) \xrightarrow{w} (\downarrow, \varepsilon)$ resp. there is some $x \in \mathcal{D}^*$ with $(s, dx) \xrightarrow{w} (t, x)$. When we have this, the problem is solved, since, taking $V_{\uparrow\varepsilon}$ as the initial variable,

$$V_{\uparrow\varepsilon} \gtrsim v\mathbf{1} \qquad \Longleftrightarrow \qquad (\uparrow, \varepsilon) \xrightarrow{v} (\downarrow, \varepsilon) \qquad \Longleftrightarrow \qquad M \text{ accepts } v.$$

The recursive specification is as follows: for every $u \in \mathcal{S}$, variable $V_{s\varepsilon}$ has a summand $a.V_{tdu}$ for every step $s \xrightarrow{\varepsilon,a,d} t$ and V_{sdt} has a summand $a.\mathbf{1}$ for every step $s \xrightarrow{d,a,\varepsilon} t$ and for all states $u, v \in \mathcal{S}$, variable V_{sdu} has a summand $a.V_{tev} \cdot V_{vdu}$ for every step $s \xrightarrow{d,a,ed} t$. Notice that the resulting specification need not be guarded, as a may be τ.

In most cases, a lot of unproductive and unreachable variables are created in this last part, but this does not affect the language generated. □

We see that the equivalence between context-free languages and push-down languages can only be established using language equivalence as the notion of equivalence. In bisimulation equivalence, it is not the case that every push-down process can be given by a recursive specification over SA, and also conversely, not every recursive specification over SA yields a push-down process. Nevertheless, carefully analyzing the proofs above, let us see what we can salvage.

Definition 4.54. • Let M be a push-down automaton that only has push and pop transitions. We say M is *popchoice-free* if for all data element $d \in \mathcal{D}$, whenever there are transitions $s \xrightarrow{d,a,\varepsilon} t$ and $u \xrightarrow{d,b,\varepsilon} v$, then $t = v$. Thus, every time a d is popped, the resulting state is the same.

- Let E be a recursive specification over SA that is in restricted Greibach normal form (see Definition 4.39). We say E is *transparency-restricted* if for all states of its transition system denoted by a sequence of variables $X \in \mathcal{N}^*$, we have that only the *last* variable of this sequence may contain a $\mathbf{1}$ summand.

Theorem 4.55. Let E be a transparancy-restricted recursive specification over SA. Then there is a popchoice-free push-down automaton denoting the same process.

Let M be a popchoice-free push-down automaton. Then there is a transparancy-restricted recursive specification over SA denoting the same process.

Proof. For the first statement let E be a transparency-restricted recursive specification, and let $S \in \mathcal{N}$ be a name in E. We define a pushdown automaton $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ as follows:

1. The set $\mathcal{S}$ consists of the names $\mathcal{N}$, the symbol $\mathbf{1}$, an extra initial state $\uparrow$, and an extra intermediate state t.
2. The set $\mathcal{A}$ consists of all the actions occurring in E.
3. The set $\mathcal{D}$ consists of the names $\mathcal{N}$ and the symbol $\mathbf{1}$.
4. The transition relation $\rightarrow$ is defined as follows:
   (a) there is a transition $\uparrow \stackrel{\varepsilon,\tau,\mathbf{1}}{\longrightarrow} S$;
   (b) if the right-hand side of the defining equation for a name $P \in \mathcal{N}$ has a summand $a.\mathbf{1}$, then $\rightarrow$ has transitions $P \stackrel{\mathbf{1},a,\varepsilon}{\longrightarrow} \mathbf{1}$ and $P \stackrel{Q,a,\varepsilon}{\longrightarrow} Q$,
   (c) if the right-hand side of the defining equation for a name $P \in \mathcal{N}$ has a summand $a.Q$, then there are transitions $P \stackrel{d,a,Qd}{\longrightarrow} t$ and $t \stackrel{Q,\tau,\varepsilon}{\longrightarrow} Q$ $(d \in \mathcal{D})$, and
   (d) if the right-hand side of the defining equation for a name $P \in \mathcal{N}$ has a summand $a.Q \cdot R$, then there are transitions $P \stackrel{d,a,Rd}{\longrightarrow} Q$ $(d \in \mathcal{D})$.
5. The set of final states $\downarrow$ consists of $\mathbf{1}$ and all variables with a $\mathbf{1}$-summand.

We leave it to the reader to check that the resulting transition systems are branching bisimilar. Using the procedure described earlier, the set of transitions can be limited to include push and pop transitions only. The push-down automaton resulting from the procedure is popchoice-free, for an P-pop transition leads to state P.

The proof of the second statement is an adaptation of the proof above. Let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be a popchoice-free push-down automaton. We define a transparency-restricted specification E with for every state $s \in \mathcal{S}$ a variable $V_{s\varepsilon}$ and for every state s a variable V_{sdt} if M has (one or more) transitions that pop datum d leading to the state t. The defining equations in E for these names satisfy the following:

1. The right-hand side of the defining equation for $V_{s\varepsilon}$ has
   (a) a summand $\mathbf{1}$ if, and only if, $s \downarrow$, and

   (b) a summand $a.V_{tdu} \cdot V_{u\varepsilon}$ whenever $s \xrightarrow{\varepsilon,a,d} t$ and all d-pop transitions lead to u.

2. $V_{s\varepsilon} = \mathbf{0}$ if there are no other summands.

3. The right-hand side of the defining equation for V_{sdt} has

   (a) a summand $a.\mathbf{1}$ if, and only if, $s \xrightarrow{d,a,\varepsilon} t$, and

   (b) a summand $a.V_{uev} \cdot V_{vdt}$ whenever $s \xrightarrow{d,a,ed} u$ and all e-pop transitions lead to state v.

4. $V_{sdt} = \mathbf{0}$ if V_{sdt} has no other summands.

It is easy to see that the resulting specification is transparency-restricted, and that the resulting trtansition systems are branching bisimilar. □

In Chapter 2, we showed that assuming determinism for automata does not change the family of languages generated. This is not the case in the present situation: deterministic push-down automata accept a smaller set of languages than non-deterministic ones.

First of all, we need to say what determinism for push-down automata exactly means. For this, we use Definition 2.56:

Definition 4.56. Let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be a push-down automaton. M is *deterministic* if its transition system is deterministic according to Definition 2.56.

A *deterministic push-down language* is a language accepted by a deterministic push-down automaton.

Example 4.57. The push-down automaton given in Figure 4.2 is not deterministic, as in Figure 4.3 we have $(s, 1) \xrightarrow{b} \!\!\!\!\rightarrow (t, \varepsilon)$ and $(s, 1) \xrightarrow{b} \!\!\!\!\rightarrow (\downarrow, \varepsilon)$. However, the push-down automaton in Figure 4.4 is deterministic. Thus, the language $\{a^n b^n \mid n \geq 0\}$ is a deterministic push-down language. The push-down automaton given in Figure 4.5 is also not deterministic, as we have $(\uparrow, a) \xrightarrow{a} \!\!\!\!\rightarrow (\uparrow, aa)$ and $(\uparrow, a) \xrightarrow{a} \!\!\!\!\rightarrow (\downarrow, \varepsilon)$. However, here the non-determinism cannot be removed (we have to 'guess' the end of the string), a deterministic push-down automaton does not exist for the language $\{ww^R\}$, this language is not deterministic push-down. We will not prove this fact.

It is easy to see that every regular language is a deterministic push-down language. Thus, we conclude that the deterministic push-down languages are a class of languages in between the regular languages and the push-down languages, and equal to neither of them.

Deterministic push-down languages have better parsing properties than general push-down languages. This can be seen by studying the class of recursive specifications that generate deterministic push-down languages. We will not pursue this matter further in this text, but just remark that all simple recursive specifications yield deterministic push-down languages (but there are deterministic context-free languages that do not have a simple recursive specification).

Exercises

4.5.1 Prove that the push-down automaton constructed in the proof of Theorem 4.52 accepts the given language L.

4.5.2 Prove that the recursive specification constructed in the proof of Theorem 4.53 generates the same language as the given push-down automaton M.

4.5.3 Prove that the push-down automaton constructed in the first half of the proof of Theorem 4.55 has a transition system that is branching bisimilar to the transition system of the given recursive specification E.

4.5.4 Prove the the recursive specification constructed in the second half of the proof of Theorem 4.55 has a transition system that is branching bisimilar to the transition system of the given push-down automaton M.

4.5.5 Construct a deterministic push-down automaton that accepts the following language:

$$L = \{wcw^R \mid w \in \{a, b\}^*\}.$$

4.5.6 Show that if L is a deterministic push-down language and L' is a regular language, then $L \cup L'$ is deterministic push-down.

4.6 Push-down processes

We investigate another characterization of push-down processes, that explicitly models the interaction in a push-down automaton between the finite control and the memory, using the tools of parallel composition, encapsulation and abstraction. The memory will be modeled by the stack process S defined in Figure 4.10.

Theorem 4.58. A process p is a push-down process if and only if there is a regular process q such that

$$p \leftrightarrows_{\mathrm{b}} \tau_{i,o}(\partial_{i,o}(q \parallel S)).$$

Proof. First of all, suppose we have a process of which the transition system consists of all executions of the push-down automaton $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$. For simplicity, assume M has push and pop transitions only. The stack S we will use has as data set the set $\mathcal{D}$ plus one extra data element $\emptyset$ that will be used to denote that the stack becomes empty, so

$$S = \mathbf{1} + \sum_{d \in \mathcal{D} \cup \{\emptyset\}} i?d.S \cdot o!d.S.$$

We define a regular process as follows: it will have variables V_{sd}, for each $s \in \mathcal{S}$ and $d \in \mathcal{D} \cup \{\emptyset\}$. The initial variable is $V_{\uparrow\emptyset}$. For each pop transition $s \xrightarrow{d,a,\varepsilon} t$, variable V_{sd} has a summand $a. \sum_{e \in \mathcal{D} \cup \{\emptyset\}} o?e.V_{te}$. For each push transition $s \xrightarrow{d,a,ed} t$, variable V_{sd} has a summand $a.i!d.V_{te}$, and for each push transition

$s \xrightarrow{\varepsilon,a,e} t$, variable $V_{s\emptyset}$ has a summand $a.i!\emptyset.V_{te}$. Whenever $s \downarrow$ every variable $V_{s\emptyset}$ has a $\mathbf{1}$ summand. Let S_x be the stack with contents x.

Now we can see that whenever the transition system has a step $(s,d) \xrightarrow{a} (t,\varepsilon)$, then

$$\partial_{i,o}(V_{sd} \parallel S_\emptyset) \xrightarrow{a} \partial_{i,o}(\sum_{e\in\mathcal{D}\cup\{\emptyset\}} o?e.V_{te}) \parallel S_\emptyset) \xrightarrow{o?\emptyset} \partial_{i,o}(V_{t\emptyset} \parallel S).$$

The abstraction operator $\tau_{i,o}()$ will rename the $o?\emptyset$-step into a τ-step that can be seen to be inert.

Second, when the transition system has a step $(s,dy) \xrightarrow{a} (t,y)$ with y nonempty, say $y = ey'$, then

$$\partial_{i,o}(V_{sd} \parallel S_y) \xrightarrow{a}\xrightarrow{o?e} \partial_{i,o}(V_{te} \parallel S_{y'}),$$

and the second step is the only step possible after a, so the resulting τ-step is inert.

Third, if $(s,dy) \xrightarrow{a} (t,edy)$ is a push transition, then

$$\partial_{i,o}(V_{sd} \parallel S_y) \xrightarrow{a}\xrightarrow{i?d} \partial_{i,o}(V_{te} \parallel S_{dy}),$$

and again the τ-step after the a-step will be inert.

Finally, for a push transition $(s,\varepsilon) \xrightarrow{a} (t,e)$, we have

$$\partial_{i,o}(V_{s\emptyset} \parallel S) \xrightarrow{a}\xrightarrow{i?\emptyset} \partial_{i,o}(V_{te} \parallel S_\emptyset).$$

For the other direction, assume we have a regular process given by a finite automaton $M = (\mathcal{S},\mathcal{A},\uparrow,\rightarrow,\downarrow)$ and a stack over finite data set $\mathcal{D}$. Now define a push-down automaton as follows:

1. The set of states is $\mathcal{S}$, the alphabet is $\mathcal{A}$, the data alphabet $\mathcal{D}$, the initial state and final states are the same;
2. Whenever $s \xrightarrow{a} t$ in M, and $a \neq i!d, o?d$, then $s \xrightarrow{d,a,d} t$ for all $d \in \mathcal{D}$ and $s \xrightarrow{\varepsilon,a,\varepsilon} t$;
3. Whenever $s \xrightarrow{i!d} t$ in M, then $s \xrightarrow{e,\tau,de} t$ for all $e \in \mathcal{D}$ and $s \xrightarrow{\varepsilon,\tau,d} t$;
4. Whenever $s \xrightarrow{o?d} t$ in M, then $s \xrightarrow{d,\tau,\varepsilon} t$.

Again we see that every move of the regular process communicating with the stack will be matched by a move in the transition system of the push-down automaton. □

Example 4.59. Consider the push-down automaton in Figure 4.13. Here, $z \in \{1,\varepsilon\}$. The resulting regular process becomes:

$$\begin{aligned}
V_{\uparrow\emptyset} &= a.i!\emptyset.V_{\uparrow 1} + c.V_{\downarrow\emptyset}\\
V_{\uparrow 1} &= a.i!1.V_{\uparrow 1} + b.(o?\emptyset.V_{\uparrow\emptyset} + o?1.V_{\uparrow 1}) + c.V_{\downarrow 1}\\
V_{\downarrow 1} &= b.(o?\emptyset.V_{\downarrow\emptyset} + o?1.V_{\downarrow 1})\\
V_{\downarrow\emptyset} &= \mathbf{1}.
\end{aligned}$$

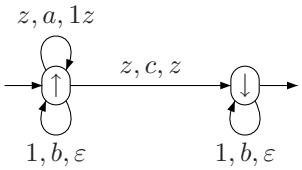

Figure 4.13: Example push-down automaton used to define regular process.

So a push-down process is a regular process communicating with a stack.

We know from Theorem 4.55 that for every transparancy-restricted recursive specification, its transition system is a push-down process. Combining this with the Theorem above, we can give a regular process that communicating with the stack yields the same process. This regular process can also be defined directly from the recursive specification.

Note 1. Let a transparancy-restricted recursive specification E over variables $\mathcal{N}$ and initial variable U be given. A regular process can be given by means of a recursive specification over MA. We use data set $\mathcal{D} = \mathcal{N} \cup \{\emptyset\}$. It will have variables V_{Pd}, with $P \in \mathcal{N}$ and $d \in \mathcal{D}$, and will communicate with a stack S over data set $\mathcal{D}$.

1. If $P \in \mathcal{N}$ has a summand $\mathbf{1}$, then $V_{P\emptyset}$ has a summand $\mathbf{1}$;
2. If $P \in \mathcal{N}$ has a summand $a.\mathbf{1}$, then $V_{P\emptyset}$ has a summand $a.\mathbf{1}$, and V_{PQ} has a summand $a.\sum_{d\in\mathcal{D}} o?d.V_{Qd}$ for all $Q \in \mathcal{N}$;
3. If $P \in \mathcal{N}$ has a summand $a.Q$, then V_{Pd} has a summand $a.V_{Qd}$ for all $d \in \mathcal{D}$;
4. If $P \in \mathcal{N}$ has a summand $a.Q \cdot R$, then V_{Pd} has a summand $a.i!d.V_{QR}$ for all $d \in \mathcal{D}$;
5. $V_{Pd} = \mathbf{0}$ in case there are no other summands.

Example 4.60. Consider the following simple example of a transparancy-restricted recursive specification:

$$\begin{aligned} U &= a.U \cdot T + c.\mathbf{1} \\ T &= b.T + d.\mathbf{1}. \end{aligned}$$

The transformation yields:

$$\begin{aligned} V_{U\emptyset} &= a.i!\emptyset.V_{UT} + c.\mathbf{1} \\ V_{UT} &= a.i!T.V_{UT} + c.(o?\emptyset.V_{T\emptyset} + o?T.V_{TT}) \\ V_{T\emptyset} &= b.V_{T\emptyset} + d.\mathbf{1} \\ V_{TT} &= b.V_{TT} + d.(o?\emptyset.V_{T\emptyset} + o?T.V_{TT} \end{aligned}$$

The other variables are $\mathbf{0}$. We show the transition systems of U and $\tau_{i,o}(\partial_{i,o}(V_{U\emptyset} \parallel S))$ in Figure 4.14.

We conclude that the stack is the prototypical push-down process, as any other push-down process can be written as a regular process communicating with the stack.

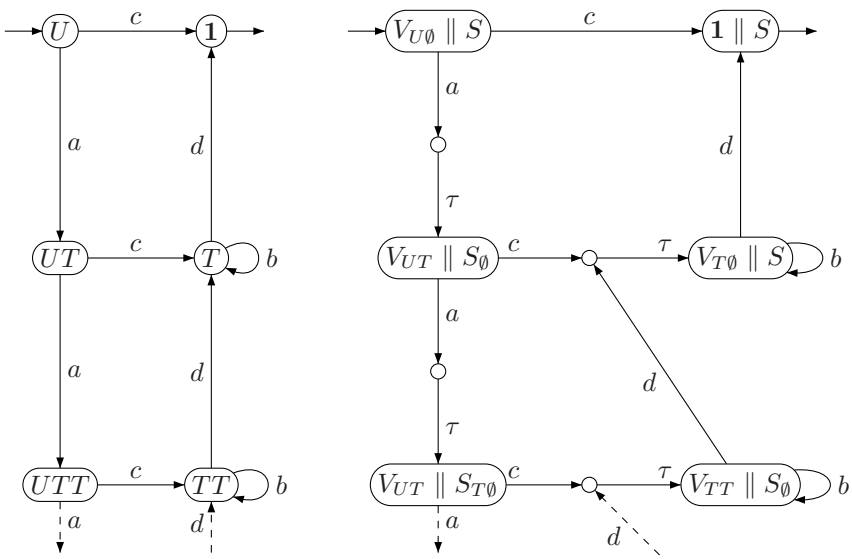

Figure 4.14: Example of a push-down process.

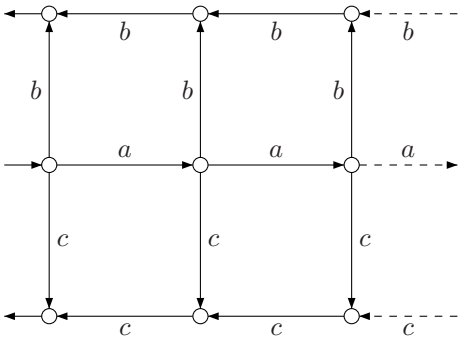

Figure 4.15: Push-down process that does not have a recursive specifitation over SA.

Exercises

4.6.1 Do the construction of Note 1 for the process given by the recursive specification $S = a.T \cdot S + b.\mathbf{1}$ and $T = c.S + d.\mathbf{1}$.

4.6.2 Do the construction of Note 1 for the stack itself given by the recursive specification

$$\begin{aligned} S &= \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.T_d \cdot S \\ T_d &= o!d.\mathbf{1} + \sum_{e \in \mathcal{D}} i?e.T_e \cdot T_d. \end{aligned}$$

4.6.3 For the process in Figure 4.15, provide a push-down automaton. Argue there is no recursive specification over SA for it.

4.7 Identifying languages that are not push-down

In this section, we will formulate and prove a pumping lemma for push-down (or context-free) languages. This lemma can be used to show that certain languages are not push-down. As was the case for the pumping lemma for regular languages, this lemma only works for infinite languages.

Theorem 4.61 (Pumping lemma for push-down languages)**.** Let L be an infinite push-down language. Then there is a positive integer m such that any $w \in L$ with $|w| \geq m$ can be written as $w = uvxyz$ with $|vxy| \leq m$ and $|vy| \geq 1$ and for all $i \geq 0$ we have that $uv^ixy^iz \in L$.

Proof. As every push-down language is context-free, we can assume E to be a recursive specification over SA with names $\mathcal{N}$ and initial variable $S \in \mathcal{N}$ such that $L = \mathcal{L}(S)$. We can assume that E does not have any $\mathbf{1}$-summands or single variable summands.

Suppose E has k equations, so $|\mathcal{N}| = k$. As E has only finitely many equations, there must be some number n such that every sequential summand has at most n symbols (in $\mathcal{A} \cup \mathcal{N}$). Since L is an infinite language, we can take a string $w \in L$ of length greater than $k * n$. Now consider a derivation $S \gtrsim w\mathbf{1}$. Each sequential term in this derivation is obtained by replacing a variable with one of its summands, so each of these terms can grow by at most $n-1$ symbols. Given the length of w, there must be a variable A that is expanded at least two times in this derivation. Take the *last* such repetition in the derivation.

For the second expansion, there must be a substring x of w such that $A \gtrsim x\mathbf{1}$. But then, considering the first expansion, there must be substrings v, y of w such that $A \gtrsim vA \cdot y\mathbf{1}$. Repeating this, we find substrings u, z with $S \gtrsim uA \cdot z\mathbf{1}$. Putting everything together, we have

$$S \gtrsim uA \cdot z\mathbf{1} \gtrsim uvA \cdot yz\mathbf{1} \gtrsim uvxyz\mathbf{1} = w\mathbf{1}.$$

By repeating the second part of this derivation i times, we see that we obtain $uv^ixy^iz \in L$. As we have taken the last repetition, there is no repeating variable in $A \gtrsim vA \cdot y\mathbf{1} \gtrsim vxy\mathbf{1}$ apart from A. This means $|vxy|$ must be bounded, we can assume $|vxy| \leq m$. As E has no $\mathbf{1}$-summands and no single variable summands, v and y cannot both be empty, so $|vy| \geq 1$. □

We see that if we have an infinite push-down language L, then if we take a string in L that is sufficiently long, then we can break this string into five parts, and the parts to the left and right of the middle part of the string can be pumped arbitrarily many times, staying inside L.

Example 4.62. $L = \{a^n b^n c^n \mid n \geq 0\}$ is not push-down. For, if L were push-down, then the pumping lemma can be applied. Take the value m given by the pumping lemma. Take string $w = a^m b^m c^m \in L$. Then write $w = uvxyz$ as given by the pumping lemma. Taking $i = 0$, we find that since $uxz \in L$, the number of a's, b's and c's in uxz must be the same. But then also the number of a's, b's and c's in vy must be the same. This number must be at least 1 by $|vy| \geq 1$. But by $|vxy| \leq m$, vy cannot contain both an a and a c. This is a contradiction. So the assumption that L was push-down was wrong. Thus, L is not push-down.

Example 4.63. $L = \{ww \mid w \in \{a, b\}^*\}$ is not push-down. For, if L were push-down, then the pumping lemma can be applied. Take the value m given by the pumping lemma. Choose string $w = a^m b^m a^m b^m \in L$. Then write $w = uvxyz$ as given by the pumping lemma. Of the four parts of which w consists, the substring vxy can only be a substring of at most two adjacent ones, since $|vxy| \leq m$. Since $|vy| \geq 1$, vy contains at least one a or at least one b. Taking $i = 0$, we have that $uxz \in L$, but the string uxz will either lack at least one a in one of the two a-groups or will lack at least one b in one of the two b-groups. This is a contradiction. So the assumption that L was push-down was wrong. Thus, L is not push-down.

Exercises

4.7.1 Show that the language $\{a^n b^n c^m \mid n \neq m\}$ is not push-down.

4.7.2 Show that the language $\{ww^R w \mid w \in \{a, b\}^*\}$ is not push-down.

4.7.3 Show that the following languages on $\mathcal{A} = \{a, b, c\}$ are not push-down:

(a) $\{a^n b^m c^k \mid k = mn\}$;
(b) $\{a^n b^m c^k \mid k > m, k > n\}$;
(c) $\{w \mid \#_a(w) < \#_b(w) < \#_c(w)\}$;
(d) $\{a^n b^m \mid n$ and m are both prime numbers $\}$.

4.7.4 Show that the following languages are not push-down:

(a) $L = \{a^{n^2} \mid n \geq 0\}$;
(b) $L = \{a^n b^m a^n \mid n, m \geq 0, n \geq m\}$;
(c) $L = \{wcw \mid w \in \{a, b\}^*\}$;
(d) $L = \{a^{m!} \mid m \geq 0\}$;
(e) $L = \{a^m b^n \mid m, n \geq 0, m = n^2\}$.

4.7.5 Determine whether or not the following languages are push-down:

(a) $\{a^n b^m a^n b^m \mid n, m \geq 0\}$;

(b) $\{a^n b^m a^m b^n \mid n, m \geq 0\}$;

(c) $\{a^n b^m a^k b^j \mid n + m \leq k + j\}$;

(d) $\{a^n b^n c^m \mid n \leq m\}$.

4.7.6 Determine whether or not the following language is push-down:

$$L = \{vcw \mid v, w \in \{a, b\}^*, v \neq w\}.$$

4.8 Properties of push-down languages and processes

We look at some properties of the class of push-down languages and the class of push-down processes and the class of processes defined by a recursive specification over SA. These properties can be used to show that a language or process is push-down or not.

Theorem 4.64. The class of processes defined by a recursive specification over SA is closed under $+, \cdot, *$ and $\partial_i()$.

Proof. Let p, q be two context-free processes. Let E and E' be two recursive specifications over SA, with initial variables respectively S and S', defining p, q.

1. Consider the recursive specification that consists of all the equations in E and E' plus the extra equation $\widehat{S} = S + S'$, with $\widehat{S}$ as initial variable. This specification defines $p + q$ and is guarded if E and E' are.

2. Consider the recursive specification that consists of all the equations in E and E' plus the extra equation $\widehat{S} = S \cdot S'$, with $\widehat{S}$ as initial variable. This specification defines $p \cdot q$ and is guarded if E is.

3. Consider the recursive specification that consists of all the equations in E plus the extra equation $\widehat{S} = \mathbf{1} + S \cdot \widehat{S}$, with $\widehat{S}$ as initial variable. This specification defines p^* and is guarded if E is.

4. For each variable P of E, let $\widehat{P}$ denote its encapsulation. Then, from E we obtain a recursive specification over variables $\widehat{P}$ by using the laws of Table 15 plus the additional law $\partial_i(x \cdot y) \leftrightarrows \partial_i(x) \cdot \partial_i(y)$. This specification defines $\partial_i(p)$ and is guarded if E is.

□

An immediate consequence of this proof is the following theorem. If we define encapsulation also on languages, we find that the class of context-free languages is also closed under encapsulation.

Theorem 4.65. The class of context-free languages is closed under union, concatenation and iteration.

Proof. As in the first three items of the proof above, using that $\mathcal{L}(\widehat{S}) = \mathcal{L}(S) \cup \mathcal{L}(S')$, $\mathcal{L}(\widehat{S}) = \mathcal{L}(S) \cdot \mathcal{L}(S')$ and $\mathcal{L}(\widehat{S}) = (\mathcal{L}(S))^*$, respectively. □

Note 2. In order to get this result for the class of push-down processes, we need to define these operators on arbitrary transition systems. This is not difficult to do, along the lines of Figures 2.29 and 3.1. Then, closure follows straightforwardly.

Theorem 4.66. The class of context-free languages is closed under abstraction.

Proof. Suppose language L is given by a recursive specification E over SA with added names $\mathcal{N}$. For each $P \in \mathcal{N}$, let $\widehat{P}$ denote its abstraction. Then, from E we obtain a recursive specification over $\{\widehat{P} \mid P \in \mathcal{N}\}$ by using the laws of Table 16 plus the additional law $\tau_i(x \cdot y) \leftrightarrows \tau_i(x) \cdot \tau_i(y)$. □

Theorem 4.67. The class of processes defined by a recursive specification over SA is closed under abstraction. However, the class of processes defined by a guarded recursive specification over SA is not.

This last fact is because the abstraction of a guarded recursive specification need not be guarded.

Theorem 4.68. The class of processes defined by a recursive specification over SA and the class of context-free languages are not closed under parallel composition.

Proof. Use Figure 4.15. □

Again, we can also define parallel composition on arbitrary transition systems. Then, it can be establsihed that also the class of push-down processes is not closed under parallel composition.

Theorem 4.69. The class of context-free languages is not closed under intersection and complementation.

Proof. Since we have to prove that something is not true, it is sufficient to give a counterexample. Consider the two languages $L = \{a^n b^n c^m \mid n, m \geq 0\}$ and $L' = \{a^n b^m c^m \mid n, m \geq 0\}$. It is not difficult to show that L and L' are context-free. For instance, a recursive specification for L is

$$\begin{aligned} S &= T \cdot U \\ T &= \mathbf{1} + a.T \cdot b.\mathbf{1} \\ U &= \mathbf{1} + c.U. \end{aligned}$$

We see $L \cap L' = \{a^n b^n c^n \mid n \geq 0\}$, and we know that this language is not context-free by Example 4.62. Thus, context-free languages are not closed under intersection.

The second part of this theorem follows from Theorem 4.65 and the identity

$$L \cap L' = \overline{\overline{L} \cup \overline{L'}}.$$

For, if the class of context-free languages were closed under complementation, then the right-hand side of this identity would be a context-free language for any context-free L, L'. But then we would have that context-free languages are closed under intersection, which is not true. □

So we see that some properties that hold for regular languages do not hold for context-free languages. The intersection of two context-free languages need not be context-free, but we do have the following.

Theorem 4.70. Let L be a context-free language and L' a regular language. Then $L \cap L'$ is context-free.

Proof. Take $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ to be a push-down automaton accepting L and take $M' = (\mathcal{S}', \mathcal{A}, \rightarrow', \uparrow', \downarrow')$ to be a deterministic automaton accepting L'. Now define a push-down automaton $\widehat{M}$ as follows:

1. $\widehat{\mathcal{A}} = \mathcal{A}$ and $\widehat{\mathcal{D}} = \mathcal{D}$;
2. $\widehat{\mathcal{S}} = \mathcal{S} \times \mathcal{S}'$;
3. $\widehat{\uparrow} = (\uparrow, \uparrow')$;
4. $\widehat{\downarrow} = \downarrow \times \downarrow'$;
5. $(s, s') \widehat{\xrightarrow{d,a,x}} (t, t')$ holds iff $s \xrightarrow{d,a,x} t$ and $s' \xrightarrow{a}{}' t'$ $(a \in \mathcal{A})$;
6. $(s, s') \widehat{\xrightarrow{d,\tau,x}} (t, s')$ holds iff $s \xrightarrow{d,\tau,x} t$.

It is not difficult to see that $\widehat{M}$ is a push-down automaton that accepts $L \cap L'$. □

The result of this theorem can be phrased as follows: the class of context-free languages is closed under regular intersection. We show how this theorem can be used in the following example.

Example 4.71. The language $L = \{w \in \{a, b, c\}^* \mid \#_a(w) = \#_b(w) = \#_c(w)\}$ is not context-free. This can be proved as follows. Since $(a.\mathbf{1})^* \cdot (b.\mathbf{1})^* \cdot (c.\mathbf{1})^*$ is an iteration expression, the language $\mathcal{L}((a.\mathbf{1})^* \cdot (b.\mathbf{1})^* \cdot (c.\mathbf{1})^*)$ is regular. Then, if L would be context-free, then also

$$L \cap \mathcal{L}((a.\mathbf{1})^* \cdot (b.\mathbf{1})^* \cdot (c.\mathbf{1})^*) = \{a^n b^n c^n \mid n \geq 0\}$$

would be context-free. This is a contradiction in view of Example 4.62.

Exercises

4.8.1 Is the complement of the language in Example 4.71 context-free?

4.8.2 Show that the family of context-free languages is closed under reversal.

4.8.3 Show that the family of context-free languages is not closed under difference. On the other hand, show that if L is context-free and L' is regular, then $L - L'$ is context-free.

4.8.4 Show that if L is deterministic context-free and L' is regular, then $L - L'$ is deterministic context-free.

4.8.5 Show that the family of deterministic context-free languages is not closed under union and not closed under intersection.

4.8.6 Give an example of a context-free language whose complement is not context-free.

4.8.7 Show that the family of unambiguous context-free languages is not closed under union and not closed under intersection.

Chapter 5

Computability and Executability

We have seen that a computer with a stack-like memory can do a number of things. The class of context-free languages can be described with such a computer. However, we have also seen some simple languages that are not context-free, in Examples 4.62 and 4.63.

In this chapter, we consider the Turing machine of Figure 1.3, reproduced here in Figure 5.1. This model can be used to define when a language is accepted by a Turing machine, but also, if we take the input symbols one at a time, describe the process of a Turing machine.

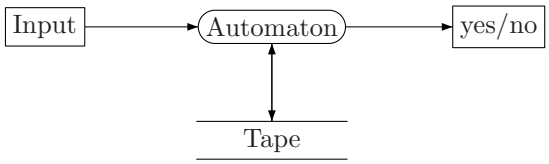

Figure 5.1: Turing machine.

The only difference with the push-down automaton of the previous chapter is that the memory which could only be accessed at the top now is replaced by a memory which again always has a string as contents, but which now can be accessed at a given point. This entails that the access point (the head of a reading device) can move around the string which is the memory content.

The Turing machine seems like a small advancement over the possibilities of a push-down automaton. Nevertheless, we will argue that any computation that can be done by any computer can also be done by a Turing machine, and we will say that a language is *computable* exactly when it is accepted by a Turing machine, and that is process is *executable* exactly when it is the process of a Turing machine.

We will generalize the case of output of only yes or no to the case where the output is an arbitrary string at the end of a computation. In this case, we speak of a *computable function*. If input and output can be taken one element

at a time, and in arbitrary order, we have to designate an input port i and an output port o. See Figure 5.2.

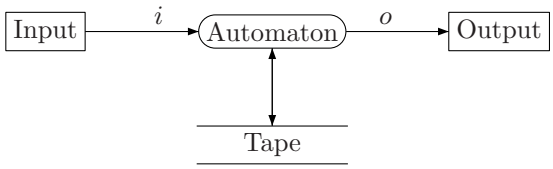

Figure 5.2: Interactive Turing machine.

5.1 The Turing machine

Definition 5.1 (Turing machine)**.** An *(Interactive) Turing machine* M is defined as a sixtuple $(\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ where:

1. $\mathcal{S}$ is a finite set of states,
2. $\mathcal{A}$ is a finite alphabet,
3. $\mathcal{D}$ is a finite set of data,
4. $\rightarrow \subseteq \mathcal{S} \times (\mathcal{D} \cup \{\varepsilon\}) \times (\mathcal{A} \cup \{\tau\}) \times (\mathcal{D} \cup \{\varepsilon\}) \times \{L, R\} \times \mathcal{S}$ is a finite set of *transitions* or *steps*,
5. $\uparrow \in \mathcal{S}$ is the initial state,
6. $\downarrow \subseteq \mathcal{S}$ is the set of final states.

If $(s, d, a, e, M, t) \in \rightarrow$, we write $s \xrightarrow{d,a,e,M} t$, and this means that the machine, when it is in state s and reading symbol d on the tape, will execute input action a, change the symbol on the tape to e, will move one step left if $M = L$ and right if $M = R$ and thereby move to state t. It is also possible that d and/or e is ε: if d is ε, we are looking at an empty part of the tape, but, if the tape is nonempty, then there is a symbol immediately to the right or to the left; if e is ε, then a symbol will be erased, but this can only happen at an end of the memory string. The exact definitions are given below.

At the start of a Turing machine execution, we will assume the Turing machine is in the initial state, and that the memory tape is empty (denoted by the *blank* symbol $\Box$).

By looking at all possible executions, we can define the transition system of a Turing machine. To define a configuration, we need to know the contents of the memory tape, a string $x \in \mathcal{D}^*$, but also the present location. This location can be an element of the memory string, but can also be immediately to the left of the memory string, or immediately to the right of the memory string. We indicate the present location by means of a bar, so if for instance 001 is the contents of the tape, then we have configurations of a state together with $\bar{\Box}001\Box$, $\Box\bar{0}01\Box$, $\Box0\bar{0}1\Box$,$\Box00\bar{1}\Box$ or $\Box001\bar{\Box}$.

Definition 5.2. Let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be a Turing machine. The *transition system* of M is defined as follows:

1. The set of states is $\{(s, \bar{\square}) \mid s \in \mathcal{S}\} \cup \{(s, \square x \square, \bar{}) \mid s \in \mathcal{S}, x \in \mathcal{D}^* - \{\varepsilon\}\}$ (in the second component there is an overbar on one of the elements of the string $\square x \square$).

2. A symbol can be replaced by another symbol if the present location is not a blank. Moving right, there are two cases: there is another symbol to the right or there is a blank to the right.

   - $(s, \square x \bar{d} \square) \xrightarrow{a} (t, \square x e \bar{\square})$ iff $s \overset{d,a,e,R}{\longrightarrow} t$ $(d, e \in \mathcal{D}, x \in \mathcal{D}^*)$,
   - $(s, \square x \bar{d} f y \square) \xrightarrow{a} (t, \square x e \bar{f} y \square)$ iff $s \overset{d,a,e,R}{\longrightarrow} t$, for all $d, e \in \mathcal{D}, x, y \in \mathcal{D}^*$.

   Similarly, there are two cases for a move left.

   - $(s, \square \bar{d} x \square) \xrightarrow{a} (t, \bar{\square} e x \square)$ iff $s \overset{d,a,e,L}{\longrightarrow} t$ $(d, e \in \mathcal{D}, x \in \mathcal{D}^*)$,
   - $(s, \square x f \bar{d} y \square) \xrightarrow{a} (t, \square x \bar{f} e y \square)$ iff $s \overset{d,a,e,L}{\longrightarrow} t$, for all $d, e \in \mathcal{D}, x, y \in \mathcal{D}^*$.

3. To erase a symbol, it must be at the end of the string. For a move right, there are three cases.

   - $(s, \square \bar{d} \square) \xrightarrow{a} (t, \bar{\square})$ iff $s \overset{d,a,\varepsilon,R}{\longrightarrow} t$ $(d \in \mathcal{D})$,
   - $(s, \square x \bar{d} \square) \xrightarrow{a} (t, \square x \bar{\square})$ iff $s \overset{d,a,\varepsilon,R}{\longrightarrow} t$ $(d \in \mathcal{D}, x \in \mathcal{D}^* - \{\varepsilon\})$,
   - $(s, \square \bar{d} f x \square) \xrightarrow{a} (t, \square \bar{f} x \square)$ iff $s \overset{d,a,\varepsilon,R}{\longrightarrow} t$ $(d \in \mathcal{D}, x \in \mathcal{D}^*)$.

   Similarly for a move left.

   - $(s, \square \bar{d} \square) \xrightarrow{a} (t, \bar{\square})$ iff $s \overset{d,a,\varepsilon,L}{\longrightarrow} t$ $(d \in \mathcal{D})$,
   - $(s, \square \bar{d} x \square) \xrightarrow{a} (t, \bar{\square} x \square)$ iff $s \overset{d,a,\varepsilon,L}{\longrightarrow} t$ $(d \in \mathcal{D}, \delta \in \mathcal{D}^* - \{\varepsilon\})$,
   - $(s, \square x f \bar{d} \square) \xrightarrow{a} (t, \square x \bar{f} \square)$ iff $s \overset{d,a,\varepsilon,L}{\longrightarrow} t$ $(d \in \mathcal{D}, x \in \mathcal{D}^*)$.

4. To insert a new symbol, we must be looking at a blank. We can only move right, if we are to the left of a (possible) data string. This means there are two cases for a move right.

   - $(s, \bar{\square}) \xrightarrow{a} (t, \square d \bar{\square})$ iff $s \overset{\varepsilon,a,d,R}{\longrightarrow} t$ $(d \in \mathcal{D})$,
   - $(s, \bar{\square} f x \square) \xrightarrow{a} (t, \square d \bar{f} x \square)$ iff $s \overset{\varepsilon,a,d,R}{\longrightarrow} t$ $(d \in \mathcal{D}, x \in \mathcal{D}^*)$.

   Similarly for a move left.

   - $(s, \bar{\square}) \xrightarrow{a} (t, \bar{\square} d \square)$ iff $s \overset{\varepsilon,a,d,L}{\longrightarrow} t$ $(d \in \mathcal{D})$,
   - $(s, \square x f \bar{\square}) \xrightarrow{a} (t, \square x \bar{f} d \square)$ iff $s \overset{\varepsilon,a,d,L}{\longrightarrow} t$ $(d \in \mathcal{D}, \delta \in \mathcal{D}^*)$.

5. Finally, looking at a blank, we can keep it a blank. Two cases for a move right.

   - $(s, \bar{\square}) \xrightarrow{a} (t, \bar{\square})$ iff $s \overset{\varepsilon,a,\varepsilon,R}{\longrightarrow} t$,

- $(s, \bar{\Box} f x \Box) \xrightarrow{a} (t, \Box \bar{f} x \Box)$ iff $s \xrightarrow{\varepsilon, a, \varepsilon, R} t$ $(x \in \mathcal{D}^*)$.

Similarly for a move left.

- $(s, \bar{\Box}) \xrightarrow{a} (t, \bar{\Box})$ iff $s \xrightarrow{\varepsilon, a, \varepsilon, L} t$,
- $(s, \Box x f \bar{\Box}) \xrightarrow{a} (t, \Box x \bar{f} \Box)$ iff $s \xrightarrow{\varepsilon, a, \varepsilon, L} t$ $(x \in \mathcal{D}^*)$.

6. The initial state is $(\uparrow, \bar{\Box})$;
7. $(s, \bar{\Box}) \downarrow$ iff $s \downarrow$.

Now we define an *executable process* as the branching bisimulation equivalence class of a transition system of a Turing machine, and a *computable language* as the language equivalence class of a transition system of a Turing machine.

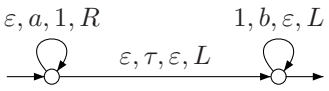

Figure 5.3: Simple Turing machine.

Example 5.3. Consider the simple Turing machine depicted in Figure 5.3. When started, any number of a's can be executed, each time writing a 1 on the tape. At any time, a move to the second state can occur, reversing direction. Then, the same number of b's can take place, each time erasing a 1. Coming to the end of the string of 1's, the tape is empty again, and termination can take place.

We depict the transition system of this Turing machine in Figure 5.4. Observe its language is $\{a^n b^n \mid n \geq 0\}$.

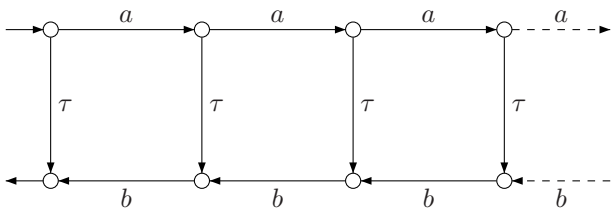

Figure 5.4: Transition system of simple Turing machine.

The language of a Turing machine can be defined directly.

Definition 5.4. Let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be a Turing machine. Then the language accepted by M, $\mathcal{L}(M)$, is defined as follows: $w \in \mathcal{L}(M)$ iff $(\uparrow, \bar{\Box}) \xrightarrow{w} \!\!\!\!\rightarrow (s, \bar{\Box})$ for some s with $s \downarrow$.

For a number of languages, Turing machines can be constructed. We start out easy.

Example 5.5. Construction of a Turing machine that accepts the language given by the iteration expression $(a.\mathbf{1})^*$ is shown in Figure 5.5. As long as the input consists of a's, we move to the right. No memory use is necessary. At any time, termination can occur. No input of a symbol different from a can occur.

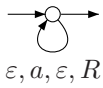

Figure 5.5: Turing machine for $\mathcal{L}((a.\mathbf{1})^*)$.

Example 5.6. The language $\{a^n b^n c^n \mid n \geq 0\}$ is not push-down. However, looking at the simple Turing machine of Figure 5.4, a Turing machine can be constructed quite easily. See Figure 5.6. This implies that a Turing machine is more powerful than a push-down automaton.

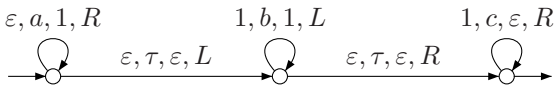

Figure 5.6: Turing machine for $\{a^n b^n c^n \mid n \geq 0\}$.

Example 5.7. With a variation on the Turing machine in Figure 5.6, also a Turing machine for the language $\{ww \mid w \in \{a, b\}^*\}$ can be constructed. Use $\mathcal{D} = \mathcal{A}$.

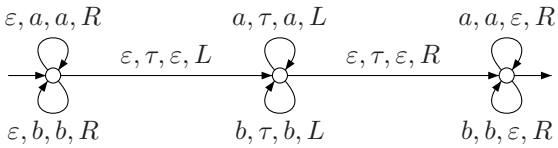

Figure 5.7: Turing machine for $\{ww \mid w \in \{a, b\}^*\}$.

Just like we did in the case of the push-down automaton (see Example 4.8), we can explicitly denote inputs and outputs of a Turing machine. Let us give the example of the (first-in first-out) queue.

Example 5.8. The queue over a data set $\mathcal{D}$ has the initial and final state at the head of the queue. There, output of the value at the head can be given, after which one move to the left occurs. If an input comes, then the position travels to the left until a free position is reached, where the value input is stored, after which the position travels to the right until the head is reached again. We show

the Turing machine in Figure 5.8 in case $\mathcal{D} = \{0, 1\}$. A label containing an n, like n, τ, n, L means there are two labels $0, \tau, 0, L$ and $1, \tau, 1, L$.

Without proof, we mention the fact that the queue process is not a push-down process.

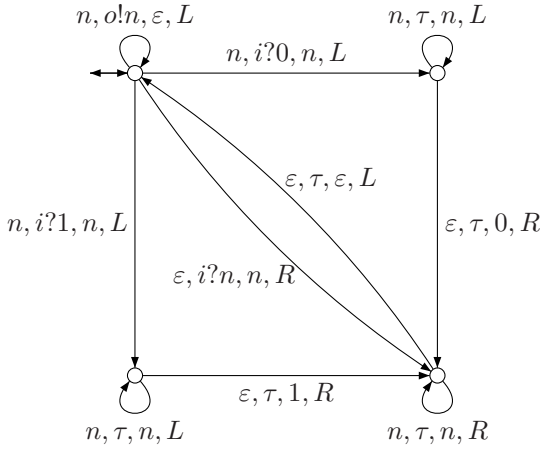

Figure 5.8: Turing machine for the queue.

Definition 5.9. We speak of a Turing machine *computation* if

1. all steps are either an input $i?d$ or an input $o!d$ or τ ($d \in \mathcal{D}$);
2. in every path from the initial state to a final state, all inputs take place before all outputs;
3. in every path from the initial state to a final state, for a sequence of inputs $i?d_1 \cdots i?d_n$ there is exactly one sequence of outputs $o!e_1 \cdots o!e_m$.

In this case, we say the Turing machine *computes* the function f on a domain of data strings D ($D \subseteq \mathcal{D}^*$) if for all input w in D it has output $f(w)$.

We will argue that for all common mathematical functions, there is a Turing machine that computes it.

Example 5.10. There is a Turing machine computing addition. For simplicity, we assume the two numbers have an equal number of digits, by adding leading 0's to the shortest number if necessary. First, the first number is input. When the + is input, the machine travels to the left until the first digit is reached again, and the second number is input, adding the two digits as we go along. When the end of the second number is reached, the machine travels back, performing the carry as needed. When the front is reached again, the resulting number can be output, erasing the tape as we go along. We show the Turing machine in Figure 5.9 in case $\mathcal{D} = \{0, 1\}$, so the case of binary addition. The carry position is on the bottom.

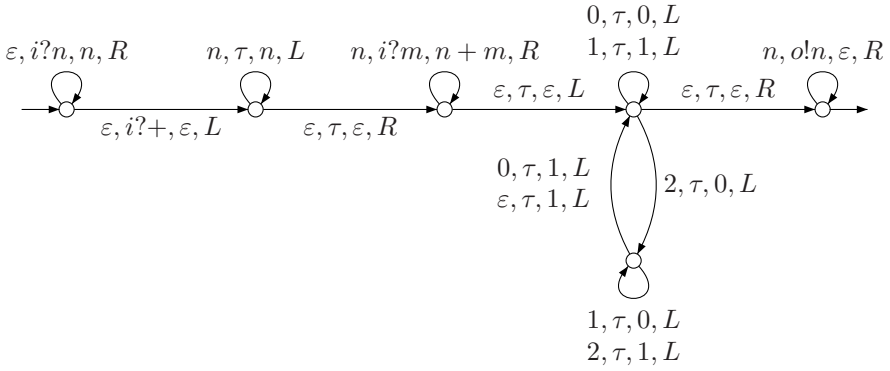

Figure 5.9: Turing machine for addition.

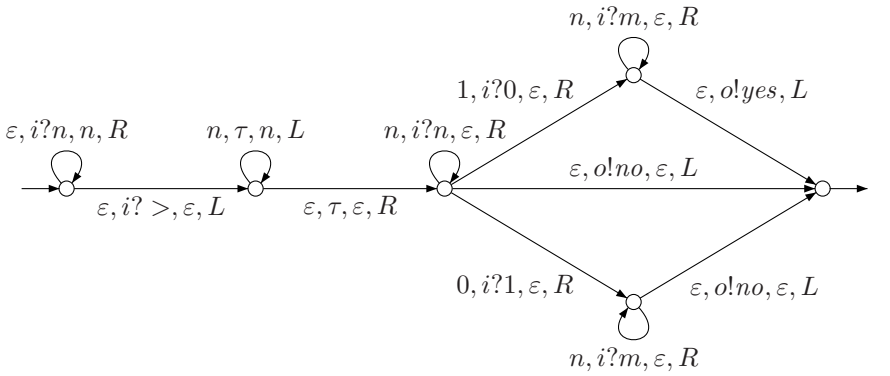

Figure 5.10: Turing machine comparing quantities.

Example 5.11. By an adaptation of the Turing machine in Figure 5.7, we can cosntruct a Turing machine computing copying, computing $f(w) = ww$ for $w \in \{a, b\}^*$. In the initial state, we change the label ε, a, a, R to $\varepsilon, i?a, a, R$ and ε, b, b, R to $\varepsilon, i?b, b, R$, in the middle state we change a, τ, a, L to $a, o!a, a, L$ and b, τ, b, L to $b, o!b, b, L$, in the final state we change a, a, ε, R to $a, o!a, \varepsilon, R$ and b, b, ε, R to $b, o!b, \varepsilon, R$.

Example 5.12. There is a Turing machine that compares quantities. Suppose we have two numbers in binary notation and we want to know whether the first number is larger than the second. As in Figure 5.9, we assume the numbers have an equal number of digits, adding leading zeroes if necessary. The input consists the two numbers separated by a $>$-sign, and $\mathcal{D} = \{0, 1, >, yes, no\}$. n, m stand for an arbitrary bit. Figure 5.10 explains the rest.

Thus, a Turing machine can be used to program a conditional split in a program. Combining the basic programs shown here, much more complicated Turing machines can be constructed, realizing more difficult program constructs.

Exercises

5.1.1 Construct a Turing machine with one state that accepts $\mathcal{L}((a.\mathbf{1}+b.\mathbf{1})^*)$, where $\mathcal{A} = \{a, b\}$.

5.1.2 Give the transition system of the Turing machine of Figure 5.6.

5.1.3 Construct a Turing machine for the language $\{a^n b^n c^n \mid n > 0\}$ that does not have any τ-moves.

5.1.4 For the Turing machine in Figure 5.7, write out the terminating path with label *aabaab* from the initial state. Give the configuration of every state passed through.

5.1.5 Give the initial part of the transition system of the queue of Figure 5.8.

5.1.6 Construct a Turing machine for binary addition, where the numbers entered need not be of equal length. Do the same for the Turing machine comparing quantities of Figure 5.10.

5.1.7 Construct Turing machines that accept the following languages over $\{a, b\}$:

(a) $\{w \mid |w| \text{ is even }\}$;

(b) $\{a^n b^m c^k \mid n \neq m \text{or} m \neq k\}$;

(c) $\{w \mid \#_a(w) = \#_b(w)\}$.

5.1.8 Construct a Turing machine that computes the function $f(w) = w^R$ on $\{a, b\}^*$.

5.1.9 Let x be a positive integer in binary notation. Construct a Turing machine that computes the function f, where $f(x) = x/2$ if x is even and $f(x) = (x + 1)/2$ if x is odd.

5.2 Church-Turing thesis

We have seen that Turing machines can accept a language that is not push-down, can define processes that are not push-down, and we have seen that Turing machines can program a number of problems that can be programmed with a computer. We now make a very sweeping statement. Any computation, any execution that can be carried out by mechanical means can be performed by a Turing machine. Thus, anything that can be done by any computer (now or in the future) can also be done by a Turing machine. This statement is called the Church-Turing thesis.

This is a statement that cannot be proved. It is like a law of nature: they can also not be proved, they can only be refuted. The Church-Turing thesis is the fundamental law of computer science.

Definition 5.13. A problem is *computable* if it can be computed by a Turing machine. An *algorithm* for a function is a Turing machine computing this function.

A process is *executable* if it is the process of a Turing machine.

Although the Church-Turing thesis cannot be proved, there are strong arguments indicating the thesis is valid.

1. Every feature of modern programming languages can be realized by means of a Turing machine;
2. No example is known of a computation or execution by computer that cannot be done by a Turing machine;
3. There are many different models of computation, and many variants on the model of a Turing machine. All of them lead to the same definition of computability and algorithm.

We present applications of these concepts.

Theorem 5.14. Let L be a regular language over alphabet $\mathcal{A}$, presented by a finite automaton, an iteration expression or a linear recursive specification, and let $w \in \mathcal{A}^*$ be a string over this alphabet. Then there is an algorithm determining whether or not $w \in L$.

Proof. If L is given by an automaton, iteration expression or recursive specification, then there is a procedure described in Chapter 2 converting this to a deterministic finite automaton. This procedure can be programmed by a Turing machine.

Given a deterministic finite automaton, we can trace w through this automaton symbol by symbol. At each point, there is exactly one edge that can be taken. Getting to the end of w, we only have to check if the resulting state is a final state or not. Again, this can be done by means of a Turing machine. □

In the proof, we presented a *membership algorithm* for regular languages.

Theorem 5.15. Let p, q be regular processes, presented by a finite automaton or a linear recursive specification. Then there is an algorithm determining whether or not $p \leftrightarrows q$.

Proof. We have not focused on procedures determining bisimulation or branching bisimulation. Nevertheless, several such procedures exist in literature. Turing machines can be constructed for them. □

Theorem 5.16. Let L be a regular language, presented by an automaton, an iteration expression or a linear recursive specification. There is an algorithm that determines whether L is empty, finite or infinite.

Proof. Again, present the language by means of a deterministic finite automaton. Reachability of states is computable. The language is empty if no final state is reachable. The language is infinite if and only if there is a cycle in the automaton that is reachable, and from which a final state can be reached. □

Theorem 5.17. Let L, L' be regular languages, presented by an automaton, an iteration expression or a linear recursive specification. There is an algorithm that determines whether $L = L'$.

Proof. Define $L'' = (L \cap \overline{L'}) \cup (\overline{L} \cap L')$. L'' is regular, and we can construct a deterministic finite automaton that accepts L''. Now $L = L'$ if and only if $L'' = \emptyset$, and we can apply the previous theorem. □

Next, we look at a membership algorithm for context-free languages.

Theorem 5.18. Let L be a context-free language over $\mathcal{A}$, presented by a recursive specification over SA or a push-down automaton, and let $w \in \mathcal{A}^*$. Then there is an algorithm determining whether or not $w \in L$.

Proof. Let L and w be given. A given recursive specification over SA or a push-down automaton for L can be converted into a recursive specification that has no $\mathbf{1}$-summands or single variable summands. Then, the exhaustive search parsing algorithm will always terminate, so can be programmed by a Turing machine. □

Theorem 5.19. Let L be a context-free language, presented by a recursive specification over SA or a push-down automaton. There is an algorithm that determines whether L is empty, finite or infinite.

Proof. If the language is given by a push-down automaton, convert this to a recursive specification over SA. The language is empty if and only if the initial variable is nonproductive.

In order to determine whether or not the language is infinite, assume we have a specification without $\mathbf{1}$-summands, without single variable summands and without nonproductive or unreachable variables. Then we claim that the language is infinite if and only if there is a repeating variable A, i.e. a variable with $A \gtrsim xA \cdot y\mathbf{1}$ for some strings $x, y \in \mathcal{A}^*$, not both empty.

For, if there is no repeating variable, then the length of every derivation is bounded and the language is finite. On the other hand, if A is repeating, then since A is reachable and productive, it generates a string in the language, and this string can be pumped, and so the language is infinite.

To determine whether a repeating variable exists, we only need to check whether or not the reachability relation has a cycle. □

Next, the following theorem comes as a surprise.

Theorem 5.20. Let L, L' be context-free languages, presented by a recursive specification over SA or a push-down automaton. There is no algorithm that determines whether or not $L = L'$.

At this point, we do not know how to prove that no algorithm exists for a certain problem. We return to this at a later time.

Exercises

5.2.1 Use a search engine to find out more about the Church-Turing thesis, about the Turing machine, and about Alonzo Church and Alan Turing.

5.2.2 Suppose regular languages L, L' are given as a finite automaton, an iteration expression or a linear recursive specification.

(a) Given a string w, present an algorithm to determine whether or not $w \in L - L'$;

(b) Present an algorithm to determine whether or not $L \subseteq L'$;

(c) Present an algorithm to determine whether or not $\varepsilon \in L$;

(d) Present an algorithm to determine whether or not $L = \mathcal{A}^*$.

5.2.3 Given a recursive specification over SA with initial variable S, present an algorithm to determine whether or not $\varepsilon \in \mathcal{L}(S)$.

5.2.4 Suppose L is a context-free language, given as a recursive specification over SA, and L' is a regular language, given as a recursive specification over MA. Present an algorithm to determine whether or not $L \cap L' = \emptyset$.

5.3 Other types of Turing machines

We show the robustness of the definition of a Turing machine: if we change the definition in several ways, nevertheless the class of languages accepted remains the same.

First, we have a look at the *Classical Turing Machine*: this machine can only define computable languages and computable functions, and is unsuitable to define executable processes. The input is entered beforehand on the memory tape (with the position at the left-most symbol of the input), then the machine executes a number of unlabeled steps (reading the current position, writing a new symbol, and moving right or left), and when the machine halts in a final state, the contents of the memory tape is considered to be the output. Acceptance of a string for a Classical Turing Machine is defined by starting with this string in memory, and then executing a number of steps to a final state, where the tape need not be empty.

Theorem 5.21. The class of computable languages and functions defined by a Classical Turing Machine is the same as the class of computable languages and functions defined by an Interactive Turing Machine.

1. Turing machines with a stay-option. If, instead of requiring that the machine at every move moves right or left, we also allow that the machine stays at the same cell, we get a definition of Turing machines that accepts the same class of languages and defines the same class of processes. This can be seen by mimicking a stay-step with a step right followed by a step left.

2. Turing machines with multiple tracks on the tape. A Turing machine with n tracks can be mimicked by a standard machine, by turning the data alphabet into an alphabet where each symbol has n components.

3. Turing machines with a tape bounded on one side. A Turing machine with a tape bounded on one side can mimick a standard machine, by dividing each cell into two parts, and 'going around the corner' when we hit the edge of the tape.

4. A Turing machine with multiple tapes. A Turing machine with n tapes can be mimicked by a Turing machine with $2n$ tracks. The even-numbered tracks put a checkmark under the symbol at the previous track, where that reading head currently is.

We very briefly indicated a number of variations that lead to the same class of languages and processes. We spend a bit more space on the *Universal Turing Machine*.

Turing machines as we have presented them up to now have a fixed program, they are designed to compute one thing. This is different from computers as we know them, that can be loaded with different programs. Also, the Turing machine can be turned into a reprogrammable device, again substantiating the claim that anything that can be done by computer can also be done by Turing machine. A *universal Turing machine* is a Turing machine that, given the description of a Turing machine M, can do all computations and executions of M.

The key to the construction of a universal Turing machine is to code a description of a Turing machine as a string. We have defined a Turing machine by means of suitably labeled graphs, but these can easily be coded as a string. For instance, the LaTeX code producing the pictures has the form of a string. A universal Turing machine can now use three tapes: the first tape starts by inputting string w_M that codes the Turing machine M, the second tape will be the working tape, taking inputs to M and producing outputs of M, and the third tape will keep track of the current state of M. Thus, at any point in a computation, the third tape will hold the current state of M and the second tape the current memory contents of M, and the control unit of the universal Turing machine will consult the first tape to see what M will do in this configuration, and will adapt the second and third tape accordingly.

Exercises

5.3.1 Sketch a Turing machine that computes multiplication of numbers in binary notation. Hint: use the Turing machines for addition and copying.

5.3.2 Give a formal definition of the Classical Turing Machine. Define a Classical Turing Machine that accepts the language $\{a^n b^n \mid n \geq 0\}$.

5.3.3 Sketch the construction of a Turing machine that accepts the language $\{ww^R \mid w \in \{a, b\}^*$. Also sketch the construction of a Turing machine that accepts the complement of this language.

5.3.4 A multihead Turing machine is a Turing machine with a single tape and single control unit but with multiple independent read-write heads. Formalise the notion of a multihead Turing machine and then indicate how such a machine can be simulated by a standard Turing machine.

5.3.5 Give an explicit encoding of a Turing machine. Compute the code of a very simple Turing machine, say the one in Figure 5.3. Now write out the construction of a universal Turing machine in some more detail.

5.4 An undecidable problem

Three main classes of formal languages have been considered. The class of regular languages contain the languages that are accepted by a computer without

memory, the class of context-free languages are accepted by a computer with restricted memory, and finally the class of languages accepted by a Turing machine holds all computable languages. We have seen examples of languages that are regular, we have seen examples that are context-free but not regular, and we have seen examples that are computable but not context-free.

In this section, we give an example of a language that is not computable. We phrase this in a slightly different way, in the form of an undecidable problem.

Definition 5.22. Suppose $\mathcal{D}$ is a given alphabet, and $D \subseteq \mathcal{D}^*$. Let P be some statement that is either *true* or *false* for every element of D. We say P is *decidable* on D if the function $f : D \to \{yes, no\}$ defined by $f(w) = yes$ if $P(w) = true$ and $f(w) = no$ if $P(w) = false$ is computable.

P is *undecidable* on D if this function f is uncomputable.

Decidability is closely related to computability. A problem is decidable if a certain input always leads to an answer (positively or negatively), so the computation is bound to terminate, may not go on indefinitely.

Decidability can also be phrased in terms of a language: problem P is decidable on D if the language $\{yw \mid P(w) = true\} \cup \{nw \mid P(w) = false\}$ is computable. Thus, exhibiting an undecidable problem amounts to exhibiting an uncomputable language.

The undecidable problem we will investigate is the *halting problem*. The halting problem is the following problem: given a Turing machine M and an input w, does M terminate on w, i.e. when M is started by the input of w, is there a computation to a final state?

Theorem 5.23. The halting problem is undecidable on the set of strings over a finite alphabet.

Proof. Suppose the halting problem is decidable. Then there must be a Turing machine H that computes the halting problem, i.e. when it gets as input the code of a Turing machine M and a string w, it will execute a step $o!yes$ leading to a final state whenever M terminates on w and will halt execute a step $o!no$ leading to a final state whenever M does not terminate on w.

Given H, we construct another Turing machine H'. Machine H' will be just like H, except that step $o!yes$ does not lead to a final state but to a deadlock state, having no outgoing edges.

We see that whenever H will terminate with $o!yes$ on input $w_M w$, then H' will deadlock, and whenever H will terminate with $o!no$ on input $w_M w$, then H' will also terminate with $o!no$ on this input.

Next, we construct another Turing machine $\widehat{H}$. When given an input that is the code of a Turing machine, w_M, it will copy this input and next enter the initial state of H', so it will behave as H' on input $w_M w_M$.

Now, $\widehat{H}$ is a Turing machine, so it has a code $w_{\widehat{H}}$. What will $\widehat{H}$ do on input $w_{\widehat{H}}$? Well, $\widehat{H}$ on input $w_{\widehat{H}}$ behaves as H' on input $w_{\widehat{H}} w_{\widehat{H}}$, so it deadlocks whenever $\widehat{H}$ terminates on input $w_{\widehat{H}}$, and it terminates in a final state whenever $\widehat{H}$ does not terminate on input $w_{\widehat{H}}$. This is a contradiction, so the halting problem must be undecidable. □

Now if we want to prove that some other problem is undecidable, then we can do this by reducing the halting problem to it: we show that assuming that

the other problem is decidable leads to the conclusion that the halting problem is decidable, thereby reaching a contradiction.

Other proofs of undecidability, uncomputability or unexecutability can be given by working in another model of computation, or starting from another well-known undecidable problem: for instance, to show that equality of context-free languages is uncomputable, it is easier to reduce the so-called Post correspondence problem to it rather than the halting problem.

Exercises

5.4.1 The *state-entry problem* is as follows: given any Turing machine M, state s of M and string w, decide whether or not the state s is entered when M gets w as input. Prove that the state-entry problem is undecidable.

5.4.2 Given any Turing machine M and string w, determining whether a symbol $d \in \mathcal{D}$ is ever written when M is gets input w is an undecidable problem. Prove this.

5.4.3 Show that there is no algorithm to decide whether or not an arbitrary Turing machine terminates on all input.

5.4.4 Show there is no algorithm to decide if two Turing machines accept the same language.

5.4.5 Is the halting problem decidable for deterministic push-down automata?

5.4.6 Show that any problem is decidable on a finite domain.

5.5 Executable processes

We investigate the class of executable processes. The first statement is obvious.

Theorem 5.24. Every push-down process is executable.

Proof. Let p be a push-down process, and let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be a push-down automaton with transition system branching bisimilar to p. We can assume M has only push and pop transitions. We define a Turing machine as follows:

1. The set of states is $\mathcal{S} \cup \{s', s'', s^* \mid s \in \mathcal{S}\}$ (three extra copies of $\mathcal{S}$;
2. the alphabet, set of data, initial state and set of final states are the same;
3. whenever $s \xrightarrow{\varepsilon,a,d} t$ is a step of the push-down automaton, then $s \xrightarrow{\varepsilon,a,d,L} s^* \xrightarrow{\varepsilon,\tau,\varepsilon,R} t$ are steps in the Turing machine;
4. whenever $s \xrightarrow{d,a,\varepsilon} t$ is a step of the push-down automaton, then $s \xrightarrow{d,a,\varepsilon,R} t$ is a step of the Turing machine;
5. whenever $s \xrightarrow{d,a,ed} t$ is a step of the push-down automaton, then $s \xrightarrow{d,a,d,L} s' \xrightarrow{\varepsilon,\tau,e,L} s'' \xrightarrow{\varepsilon,\tau,\varepsilon,R} t$ are steps in the Turing machine.

We leave it to the reader to check that this Turing machine defines p. □

Theorem 5.25. The class of executable processes is closed under $a.\_, +, \cdot, \_^* \parallel , \partial_p(), \tau_p()$.

Proof. Like Note 2 in Chapter 4. □

Just like we did in Theorem 4.58, also in a Turing machine we can make the communication between control and memory explicit. In the push-down automaton, the memory was the stack process. Here, we need to define the tape process. We do this by using *two* stacks: one holding the contents of the memory to the left of the current position, one holding the contents of the memory to the right of the current position. At all times, we need to know whether the stack to the right or the one to the right is empty or not.

Thus, we have given two stack processes, the right stack S^r and the left stack S^l. Given a data set $\mathcal{D}$, we use one extra symbol $\emptyset$ to denote an empty stack.

$$\begin{aligned} S^r &= \mathbf{1} + \sum_{k\in\mathcal{D}\cup\{\emptyset\}} r?k.S^r \cdot r!k.S^r \\ S^l &= \mathbf{1} + \sum_{k\in\mathcal{D}\cup\{\emptyset\}} l?k.S^l \cdot l!k.S^l \end{aligned}$$

Next, we define a regular control process C, with variables:

1. the initial state $C_{\emptyset\varepsilon\emptyset}$: looking at a blank, both stacks are empty;
2. $C_{\emptyset d\emptyset}$: reading $d \in \mathcal{D}$, both stacks are empty;
3. $C_{\varepsilon\emptyset}$: looking at a blank, right stack is empty, left stack is not;
4. $C_{d\emptyset}$: reading $d \in \mathcal{D}$, right stack is empty, left stack is not;
5. $C_{\emptyset\varepsilon}$: looking at a blank, left stack is empty, right stack is not;
6. $C_{\emptyset d}$: reading $d \in \mathcal{D}$, left stack is empty, right stack is not;
7. C_d: reading $d \in \mathcal{D}$, neither stack is empty.

The specification of C is now as follows:

$$
\begin{aligned}
C_{\emptyset\varepsilon\emptyset} &= \mathbf{1} + i?\varepsilon.C_{\emptyset\varepsilon\emptyset} + \sum_{d\in\mathcal{D}} i?d.C_{\emptyset d\emptyset} + i?L.C_{\emptyset\varepsilon\emptyset} + i?R.C_{\emptyset\varepsilon\emptyset} + o!\varepsilon.C_{\emptyset\varepsilon\emptyset}\\
C_{\emptyset d\emptyset} &= i?\varepsilon.C_{\emptyset\varepsilon\emptyset} + \sum_{e\in\mathcal{D}} i?e.C_{\emptyset e\emptyset} + i?L.r!\emptyset.r!d.C_{\emptyset\varepsilon} + i?R.l!\emptyset.l!d.C_{\varepsilon\emptyset} + o!d.C_{\emptyset d\emptyset}\\
&\quad (d\in\mathcal{D})\\
C_{\emptyset\varepsilon} &= i?\varepsilon.C_{\emptyset\varepsilon} + \sum_{d\in\mathcal{D}} i?d.C_{\emptyset d} + i?R.\sum_{d\in\mathcal{D}} r?d.(r?\emptyset.C_{\emptyset d\emptyset} + \sum_{e\in\mathcal{D}} r?e.r!e.C_{\emptyset d})\\
&\quad + o!\varepsilon.C_{\emptyset\varepsilon}\\
C_{\emptyset d} &= i?\varepsilon.C_{\emptyset\varepsilon} + \sum_{e\in\mathcal{D}} i?e.C_{\emptyset e} + i?L.r!d.C_{\emptyset\varepsilon} +\\
&\quad + i?R.l!\emptyset.l!d.\sum_{e\in\mathcal{D}} r?e.(r?\emptyset.C_{e\emptyset} + \sum_{f\in\mathcal{D}} r?f.r!f.C_e) + o!d.C_{\emptyset d} \quad (d\in\mathcal{D})\\
C_{\varepsilon\emptyset} &= i?\varepsilon.C_{\varepsilon\emptyset} + \sum_{d\in\mathcal{D}} i?d.C_{d\emptyset} + i?L.\sum_{d\in\mathcal{D}} l?d.(l?\emptyset.C_{\emptyset d\emptyset} + \sum_{e\in\mathcal{D}} l?e.l!e.C_{d\emptyset})\\
&\quad + o!\varepsilon.C_{\varepsilon\emptyset}\\
C_{d\emptyset} &= i?\varepsilon.C_{\varepsilon\emptyset} + \sum_{d\in\mathcal{D}} i?e.C_{e\emptyset} + i?L.r!\emptyset.r!d.\sum_{e\in\mathcal{D}} l?e.(l?\emptyset.C_{e\emptyset} + \sum_{f\in\mathcal{D}} l?f.l!f.C_e)\\
&\quad + i?R.l!d.C_{\varepsilon\emptyset} + o!d.C_{d\emptyset} \quad (d\in\mathcal{D})\\
C_d &= \sum_{e\in\mathcal{D}} i?e.C_e + i?L.r!d.\sum_{e\in D} l?e.(l?\emptyset.C_{\emptyset e} + \sum_{f\in\mathcal{D}} l?f.l!f.C_e)\\
&\quad + i?R.l!d.\sum_{e\in D} r?e.(r?\emptyset.C_{e\emptyset} + \sum_{f\in\mathcal{D}} r?f.r!f.C_e) + o!d.C_d \quad (d\in\mathcal{D}).
\end{aligned}
$$

Finally, the tape process can be defined as follows:

$$Tape = \tau_{l,r}(\partial_{l,r}(S^l \parallel C_{\emptyset\varepsilon\emptyset} \parallel S^r)).$$

We see the tape process is an executable process.

Now we can prove the analogue of Theorem 4.58. In this way, we get a specification over CA for every executable process.

Theorem 5.26. A process p is an executable process if and only if there is a regular process q such that

$$p \leftrightarrows_{\mathrm{b}} \tau_{i,o}(\partial_{i,o}(q \parallel Tape)).$$

Proof. First of all, suppose we have a process of which the transition system consists of all executions of the Turing machine $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$.

We define a regular process as follows: it will have variables V_{sk}, for each $s \in \mathcal{S}$ and $k \in \mathcal{D} \cup \{\varepsilon\}$. The initial variable is $V_{\uparrow\varepsilon}$. For each transition $s \stackrel{k,a,j,R}{\longrightarrow} t$, variable V_{sk} has a summand $a.i!j.i!R.\sum_{m\in\mathcal{D}\cup\{\varepsilon\}} o?m.V_{tm}$. Likewise, for each transition $s \stackrel{k,a,j,L}{\longrightarrow} t$, variable V_{sk} has a summand $a.i!j.i!L.\sum_{m\in\mathcal{D}\cup\{\varepsilon\}} o?m.V_{tm}$. Whenever $s\downarrow$ every variable $V_{s\varepsilon}$ has a $\mathbf{1}$ summand. Now it is just a matter of checking every transition.

The other direction simply follows from the closure properties of the set of executable processes. □

We can specialize this theorem to the case of the computable function. Given a computable function $f : \mathcal{D}^* \to \mathcal{D}^*$, take a Turing machine for it, and let Q be a linear specification of the regular process of the previous theorem. Since we used communication ports i, o for the communication of the control of the Turing machine with the tape process, we will have to use different port names for the input and the output of the computable function: we will use in, out.

Theorem 5.27. Let $f : \mathcal{D}^* \to \mathcal{D}^*$ be a computable function. Then

$$f(d_1 \dots d_n) = e_1 \dots e_m \qquad \Leftrightarrow$$

$$\tau_{in}(\partial_{in}(in!d_1 \dots in!d_n.\mathbf{1} \parallel \tau_{i,o}(\partial_{i,o}(Q \parallel Tape)))) \leftrightarrows_{\mathrm{b}} out!e_1 \dots out!e_m.\mathbf{1}$$

and

$$f(d_1 \dots d_n) \text{ is not defined} \qquad \Leftrightarrow$$

$$\tau_{in}(\partial_{in}(in!d_1 \dots in!d_n.\mathbf{1} \parallel \tau_{i,o}(\partial_{i,o}(Q \parallel Tape)))) \leftrightarrows_{\mathrm{b}} \mathbf{0}.$$

Any executable process can be specified by means of a recursive specification over CA. Often, a more direct specification can be given.

As an example, we give a recursive specification of the queue of Figure 5.8. The specification is based on the principle that putting two (unbounded) queues in a row again yields a queue. For the internal, connecting port we use l. The two constituating queues can again be specified using an internal port...

$$Q^{io} = \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.\tau_l(\partial_l(Q^{il} \parallel o!d.Q^{lo}))$$
$$Q^{il} = \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.\tau_o(\partial_o(Q^{io} \parallel l!d.Q^{ol}))$$
$$Q^{lo} = \mathbf{1} + \sum_{d \in \mathcal{D}} l?d.\tau_i(\partial_i(Q^{li} \parallel o!d.Q^{io}))$$
$$Q^{ol} = \mathbf{1} + \sum_{d \in \mathcal{D}} o?d.\tau_i(\partial_i(Q^{oi} \parallel l!d.Q^{il}))$$
$$Q^{li} = \mathbf{1} + \sum_{d \in \mathcal{D}} l?d.\tau_o(\partial_o(Q^{lo} \parallel i!d.Q^{oi}))$$
$$Q^{oi} = \mathbf{1} + \sum_{d \in \mathcal{D}} o?d.\tau_l(\partial_l(Q^{ol} \parallel i!d.Q^{li}))$$

Bibliography

[1] A. Asteroth and C. Baier. *Theoretische Informatik*. Pearson, 2003.

[2] J.C.M. Baeten, T. Basten, and M.A. Reniers. *Process Algebra (Equational Theories of Communicating Processes)*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2009.

[3] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. On the consistency of Koomen's fair abstraction rule. *Theoretical Computer Science*, 51:129–176, 1987.

[4] J.C.M. Baeten, F. Corradini, and C.A. Grabmayer. A characterization of regular expressions under bisimulation. *Journal of the ACM*, 54(2):6.1–28, 2007.

[5] J.C.M. Baeten, P.J.L. Cuijpers, B. Luttik, and P.J.A. van Tilburg. A process-theoretic look at automata. In *Proceedings of FSEN 2009*, number 5961 in LNCS, pages 1–33, Berlin-Heidelberg, 2010. Springer-Verlag.

[6] J.C.M. Baeten, P.J.L. Cuijpers, and P.J.A. van Tilburg. A context-free process as a pushdown automaton. In F. van Breugel and M. Chechik, editors, *Proceedings CONCUR'08*, number 5201 in Lecture Notes in Computer Science, pages 98–113, 2008.

[7] J.C.M. Baeten, P.J.L. Cuijpers, and P.J.A. van Tilburg. A basic parallel process as a parallel pushdown automaton. In D. Gorla and T. Hildebrandt, editors, *Proceedings EXPRESS'08*, Electronic Notes in Theoretical Computer Science, 2009. To appear.

[8] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[9] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson, 2006.

[10] P. Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett, 2001.

[11] C.A. Middelburg and M.A. Reniers. *Introduction to Process Theory*. Technische Universiteit Eindhoven, 2004.

[12] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[13] F. Moller. Infinite results. In U. Montanari and V. Sassone, editors, *Proceedings CONCUR'96*, number 1119 in Lecture Notes in Computer Science, pages 195–216, 1996.

[14] E. Rich. *Automata, Computability, and Complexity.* Pearson, 2008.

[15] R. Smullyan. *The Lady or the Tiger? And Other Logic Puzzles Including a Mathematical Novel that Features Gödel's Great Discovery.* Alfred A. Knopf, Inc., New York, USA, 1982.