Parametrized Automata over Infinite Alphabets

Properties and Complementation

A Master Thesis by:

Franziska Alber

Supervised by:

Prof. Dr. Clara Löh

Prof. Dr. Philipp Rümmer

University of Regensburg

FACULTY OF MATHEMATICS

COMPUTATIONAL SCIENCE M.SC.

Regensburg, October of 2024

Contents

Chapter 1

Introduction

Finite-state automata (FSA, [11]) are a well-researched model of computation and act as acceptors of regular languages. FSA are generally pleasant to work with, as they are closed under the Boolean operations of union, intersection and complementation and important decision problems, such as universality, non-emptiness, membership and containment, are all decidable ([11, chapters 4.2 and 4.3]).

However, since FSA only have a finite number of states and a finite transition relation describing which letters permit transitions between which states, the use of FSA is limited to regular languages over finite alphabets. In order to handle infinite alphabets, several different modifications to finite-state automata have been proposed; each with their own strengths and weaknesses.

Symbolic automata ([7]) are using logical formulas instead of letters in their transitions. Algorithms on symbolic automata are often very similar to the ones for finite-state automata, as they partition an infinite alphabet into a finite number of sets whose members have equivalent properties within the automaton.

Variable automata ([10]) command a finite number of variables which can store input letters, and compare those to upcoming letters for both equality and inequality. The variables cannot be reassigned. Variable automata and symbolic automata are incomparable, as symbolic automata are incapable of comparing different letters of an input word while variable automata do not evaluate letters using logical formulas.

This thesis is concerned with parametrized automata (PA, as introduced in [13, section 3]). PA combine the power of symbolic automata and variable automata: The transitions are labeled with logical formulas, which may also contain (non-reassignable) parameters. As such, PA subsume both symbolic automata and variable automata, but also inherit some of their weaknesses.

Up to this point, to our knowledge there have been no works centered solely on PA. Publications on the decision properties of sequence theories ([13]) and data words ([9]) have featured PA, but only examined properties of PA that were relevant to their respective research questions. The following work aims to be the first of its kind investigating the general properties of PA, with special attention paid to the problem of complementing PA. We will investigate the closure properties of PA regarding Boolean operations, see how concepts such as ε-transitions, determinism and minimization translate to PA, and determine whether some relevant decision problems such as the universality and non-emptiness problems are decidable for PA. We will also identify useful subclasses of PA, for example SDPA which have pleasant computational properties, CFPA which are easy

to complement or strict PA which provide additional control over their parameters.

There are other classes of automata that share similarities with parametrized automata, or inhabit a similar niche. An overview can be found in section 2.3.4, which also contextualizes parametrized automata in this larger ecosystem.

1.1 Motivating Example

We will motivate our work with a brief example: Dijkstra's Self-Stabilizing Protocol, a self-stabilization algorithm for distributed systems. This motivating example is mostly relevant to readers already familiar with automata theory, as we will skip over a lot of prerequisite knowledge. Those who are new to the subject are encouraged to skip ahead to chapter 2 and revisit the example at a later point.

Example 1.1 (Dijkstra's Self-Stabilizing Protocol [8])**.** Consider a ring of processes $1, \ldots, n$. Each process i is assigned a variable $v(i) \in \{0, 1, \ldots, k-1\}$, where $k \geq n$. We say that process 1 is privileged if $v(1) = v(n)$, and process $i \neq 1$ is privileged if $v(i) \neq v(i-1)$.

In Dijkstra's Self-Stabilizing Protocol, in each step, a privileged process i is randomly chosen and updated in the following way:

- If $i = 1$, then $v(1) := v(1) + 1 \pmod{k}$.
- If $i > 1$, then $v(i) := v(i-1)$.

The remainder of processes remains unchanged.

"Self-stabilizing" means: After a sufficient number of steps, the system reaches a stable state in which exactly one process is privileged. This is remarkable because each change is performed based on local information, as each variable $v(i)$ is only compared to $v(i) - 1 \pmod{k}$. While we will not prove the self-stabilizing property, we will model and prove a partial result.

Claim 1.2. The set of all states in which exactly one processor is privileged forms an invariant under Dijkstra's Self-Stabilizing Protocol.

If exactly one process is privileged, and a step of the protocol is applied, there will still be exactly one privileged process. The protocol does not "create" additional privileged processes. In other words, once a stable state has been reached it cannot be left again.

If we want to formally prove this claim, we first need to formalize the problem.

As a first step, our system consists of n processes, each of which is assigned a variable ranging from 0 to $k-1$. Each possible state of the system can therefore be encoded by a word of length n with letters from the alphabet $\{0, 1, \ldots, k\}$. Assuming that n and k are arbitrary numbers, we want to choose an approach that works for *any* n and *any* k. Since we cannot put an upper bound on k, we desire a model of computation that can handle infinite alphabets. In this particular case, the infinite alphabet should be the set of natural numbers $\mathbb{N}$. Examples of automata classes that are designed for infinite alphabets are register automata, variable automata or symbolic automata.

Another requirement results from the need to compare different letters of a word, and apply functions and arithmetic operations to those letters. *Register automata* and *variable automata* are capable of storing previous letters of a word and compare those to upcoming letters, but are blind to the "structure" of the alphabet. *Symbolic automata* function by checking whether the input letters satisfy logical formulae, but cannot store or compare input letters. Parametrized automata come to the rescue, as they combine the methods of variable automata and symbolic automata.

Let $(v(1), v(2), \ldots, v(n))$ be a system state. Observe that, for $n \geq 2$, the system will always have at least one privileged process: If all variables $v(i)$ are equal, then process 1 has to be privileged. If not, then there has to be some position i such that $v(i) \neq v(i-1)$. We can therefore rephrase the protocol in the following way:

1. Randomly choose a position of $(v(1), v(2), \ldots, v(n))$.
2. (a) If the chosen letter $v(i)$ is the first letter and $v(1) = v(n)$, update $v(1)$ to $v(1)' = v(1) + 1 \bmod k$.
   (b) If the chosen letter $v(i)$ is not the first letter and $v(i-1) \neq v(i)$, update $v(i)$ to $v(i)' = v(i-1)$.
3. The remaining letters $v(j), j \neq i$ are set to $v(j)' = v(j)$.
4. If a letter has been updated in step 2, the protocol has been applied successfully. Return the new word $(v(1)', v(2)', \ldots, v(n)')$. Otherwise, abort.

The changes are subtle, yet point the way towards a possible approach. As it turns out, parametrized automata and their extension, parametrized transducers, are sufficiently expressive to model all of the intermediate steps outlined above. Transducers ([17]) will only occur this one time and then never again, so we refrain from a lengthy introduction. A transducer is an automaton that simultaneously generates an output word. In each transition, a letter determined by the current state, input letter and specifications in the transition label is appended to a new word that is returned at the end of an accepting run. We only need this property to update and output the transformed input word. Related concepts are Mealy machines ([15]), which need to be deterministic, and Moore machines ([3]), in which only the current state but not the current letter determine the output symbol.

The manner in which a process is updated depends on its position, so we need to differentiate between the first letter and the rest. This is no challenge for most automata classes known, since it is only a matter of arranging the states properly. By not permitting loops leading to the initial state, the transitions that can be taken by the first letter are exactly the transitions that leave the initial state. This way, we can ensure that the transformation specified in step 2. a) can only be applied the first letter, and step 2. b) is never applied to the first letter.

We can differentiate between different positions in the word without a parameter, and neither do we need a parameter to encode the length of the word n. However, the maximum $k-1$ should be encoded by a parameter. This way, the same automaton can be used for different values of k. We also need a distinct parameter to store a letter, since different letters of the input word need to be compared. If $v(1)$ is chosen for updating, $v(1)$ has to be stored so the automaton can compare it to $v(n)$ at the end and verify

whether $v(1)$ was privileged. If a different $v(i)$ is chosen, $v(i-1)$ needs to be stored, both to overwrite the old $v(i)$ and to confirm that $v(i)$ and $v(i-1)$ are distinct.

As soon as the selected letter is updated and it is verified that the corresponding process is privileged, the automaton enters an accepting state.

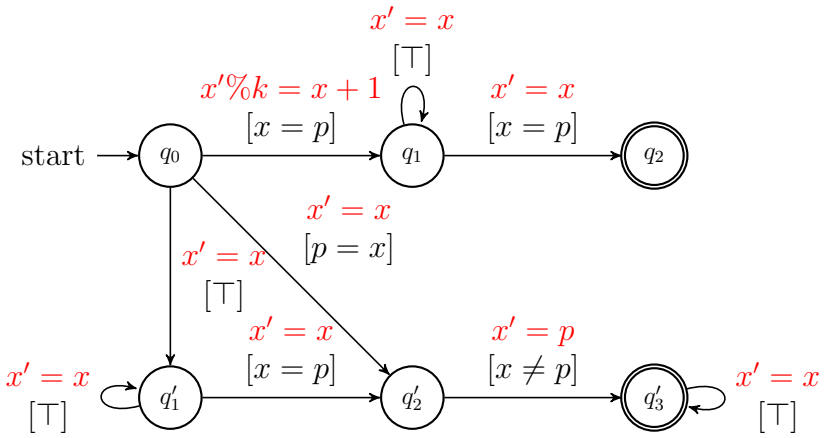


Figure 1.1: A transducer T that applies a step of Dijkstra's Self-Stabilizing Protocol to an input ring of processes.

These considerations lead to the transducer T in figure 1.1. T takes an input word $(v(1), v(2), \ldots, v(n))$, where x denotes the current letter while p and k are variables, k encoding the maximum number that can be assigned to a process. In each transition, the letter x' represents the output letter appended to the word $(v(1)', v(2)', \ldots, v(n)')$ while the input letter x has to satisfy the conditions in square brackets.

The state q_1 is reached when the first process is randomly chosen to be updated, and when variable p is randomly assigned the value $v(1)$. In this case, $v(1)'$ is set to $v(1) + 1$ mod k. All other remaining processes remain unchanged. The transducer cannot verify whether process 1 was actually privileged until the last transition, in which it confirms that $v(n) = v(1) = p$.

If a different privileged process $i \neq 1$ is chosen, the run will instead terminate in state q'_3. First, state q'_2 is entered when $p = v(i-1)$. The next transition both verifies that process i was privileged, and sets $v(i) := v(i-1) = p$.

The output word $(v(1)', v(2)', \ldots, v(n)')$ is a word to which a step of Dijkstra's Self-Stabilizing Protocol has been applied. We want to prove how, if the input word has exactly one privileged process, the output word also does. For this purpose, the property "*The word has exactly one privileged process*" should also be modeled. There are only two possible configurations for this scenario:

- If process 1 is privileged, $v(1) = v(2) = \cdots = v(n) = m$ for some $m \in \{0, 1, \ldots, k-1\}$.
- If some process $i \neq 1$ is privileged, then $v(1) = v(2) = \cdots = v(i-1) = m$ and $v(i) = v(i+1) = \cdots = v(n) = l$ for some $m, l \in \{0, 1, \ldots, k-1\}$ where $m \neq l$.

This, too, can be modeled using a parametrized automaton, seen in figure 1.2.

The automaton forces $p = v(1)$ and can terminate in state q_2 if process 1 is privileged. Otherwise, if some process $i \neq 1$ is privileged and the second variable p_2 is randomly

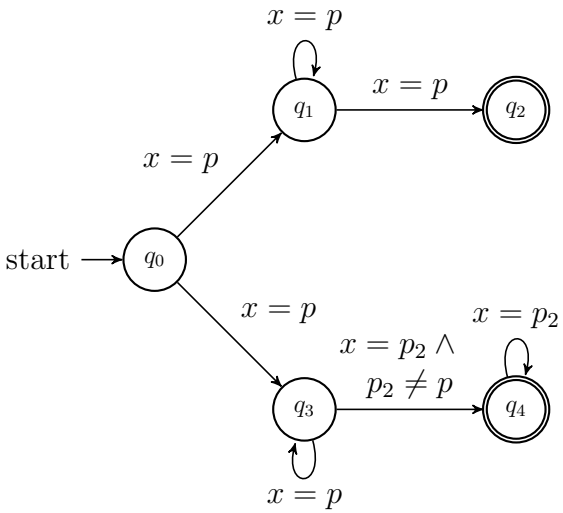


Figure 1.2: A non-deterministic parametrized automaton P that witnesses whether the input ring of processes has exactly one privileged process.

assigned the value $v(i)$, the run will terminate in q_4 if $v(1) = v(2) = \cdots = v(i-1) = p \neq p_2 = v(i) = v(i+1) = \cdots = v(n)$. The second variable is needed to ensure that no more changes to the values occur, since this would mean there is more than one privileged process.

Now that we have defined a transducer T applying a step of the protocol, and an automaton P verifying that a sequence $(v(1), \ldots, v(n))$ has exactly one privileged process, we can express claim 1.2 in the following way: If $(v(1), \ldots, v(n))$ is accepted by P, and $(v(1)', \ldots, v(n)')$ is the word resulting from applying T to $(v(1), \ldots, v(n))$, then $(v(1)', \ldots, v(n)')$ is also accepted by P. Preferably, we turn this into a non-existence statement: There is no word $(v(1), \ldots, v(n))$ that is accepted by P and T where the output word $(v(1)', \ldots, v(n)')$ is *not* accepted by P.

Here lies the problem, because in its current form, P is not suited to give information on which words it does not accept. The automaton is non-deterministic and geared towards acceptance. Both the parameters and several transitions in the automaton are chosen non-deterministically, so a single run may not terminate in an accepting state for a multitude of reasons that are hard to disentangle. We need to construct a *complement automaton* P^c that accepts exactly the words P does not accept.

One approach would be to repeat the procedure done for the construction of T and P: Reason about which words P^c should accept and then manually put together an automaton fulfilling these properties.

- P^c should accept the empty word.
- P^c should accept all words of length 1.
- P^c should accept no words of length 2, as words of length 2 always have exactly one privileged process.
- P^c should accept all words that have two or more privileged processes.

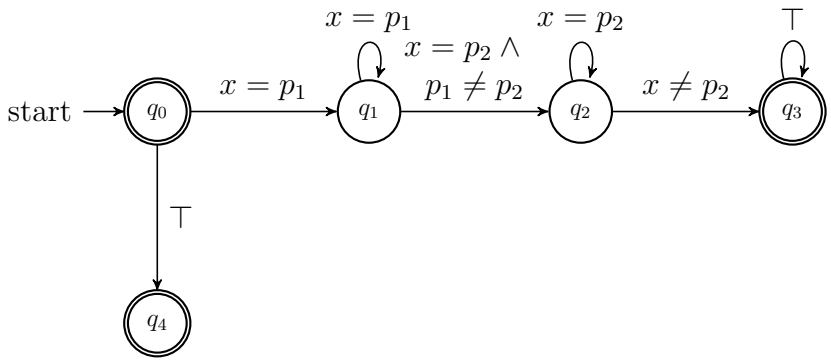


Figure 1.3: A complement automaton P^c for the parametrized automaton in figure 1.2.

The last demand can seem tricky at first, because it looks like again cases where the first process is privileged need to be differentiated from cases where it is not. However, if the first process and some other process are privileged, therefore not being accepted by P, then there automatically needs to be a third privileged process. As a consequence, an automaton P^c never needs to check if the first process is privileged. The automaton in figure 1.3 should fulfill all of these properties, although no proof is supplied.

In conclusion, the chosen procedure is expedient, but tedious, and does not come with a built-in way to verify whether P^c works correctly. Maybe we can find a more general method that works on every parametrized automaton, instead of having to hand-tailor results?

This thesis is going to explore the quirks and challenges around the complementation of parametrized automata in depth.

1.2 Contributions

Front and center in this thesis are parametrized automata, which up to this point have mostly played a side role in investigating the decision properties of sequence theories ([13]) and the study of data words ([9]). Since our ultimate goal is to find methods for complementing PA, contributions can be split into three focal points:

- This work aims to be a first port of call for readers who seek general information on PA. Therefore, we provide a formal, generalized definition of PA and study a number of important properties. Most important for the subsequent chapters is the observation that PA are not closed under complementation, and that the problem of either complementing an arbitrary PA, or returning that a complement does not exist, is undecidable. Regarding other Boolean operations on PA, we will provide algorithmic proof that PA are closed under intersection and union. We will also study classical decision problems and prove that the non-emptiness and reachability problems are decidable for PA while the universality, equivalence and containment problems are not. We will also investigate how established operations on related models of computation, such as ε-elimination, determinization, minimization and product constructions, translate to PA.

- We introduce *strongly deterministic* PA (SDPA), an approach inspired by the complementation algorithms on deterministic finite-state and symbolic automata. SDPA are a fragment of PA that is closed under all Boolean operations, is easy to complement and for which the universality, equivalence and containment problems are decidable. However, not every PA is equivalent to an SDPA, and the problem of determinizing a given PA is undecidable, limiting the worth of SDPA.

- The main contribution of this paper is the introduction of PA in *complementable normal form* (CFPA). CFPA have all the advantages of SDPA (closure under Boolean operations and decidability of universality, containment and equivalence problems) while encompassing a much wider range of PA: Not every complementable PA can be transformed into an SDPA, but it can always be transformed into a CFPA. We provide a promising method for constructing CFPA.

1.3 Outline

In order to prepare every reader for the more specific concepts encountered in this thesis while also establishing naming conventions and highlighting important definitions, chapter 2 will begin with a brief review of formal languages and first-order logic. Section 2.2 will formally define finite-state automata (FSA) and explain complementation in automata, highlighting important properties while not digging too deeply. Many of the concepts encountered in FSA will also be relevant to PA, so the section serves to both introduce these concepts and provide a reference point contextualizing the difference between finite-state automata and parametrized automata. Then in section 2.3, we will introduce multiple classes of automata for languages over infinite alphabets. Since variable automata and symbolic automata, both of which we will encounter, are related to parametrized automata, this section will prime the reader for some of the challenges we will face later.

Using the groundwork from chapter 2, chapter 3 formally defines parametrized automata. We will see very quickly that PA cannot always be complemented, and establish a few useful operations and properties in section 3.2. Section 3.3 contains algorithms for the computation of the union and intersection of PA, as well as proof that either of these always exist. In section 3.4, we will also see that the subclass of PA that are complementable is closed under the operations of union, intersection, complementation and reversal. Section 3.5 touches on the decision problems of universality, equivalence and containment (all undecidable for PA) as well as non-emptiness and reachability of states (which are decidable).

In search for a subclass of PA that is easier to complement, we introduce a notion of *strong determinism* in chapter 4. In section 4.1, we will see that strongly deterministic PA, called SDPA, have nice closure properties and can be complemented reliably and efficiently. All of the decision problems we previously considered are decidable when restricted to SDPA. Unfortunately, as shown in section 4.2, not every PA can be transformed into an SDPA, it is hard to transform a given PA into an equivalent SDPA, and even more damning, we may not even be able to verify whether a given PA is strongly deterministic.

Chapter 5 provides a brief excursion on the properties of the parameter assignment. While interesting, the chapter can be safely skipped without

Therefore, chapter 6 introduces another subclass of PA: CFPA, or parametrized automata in complementable normal form. In sections 6.1 and 6.2, plenty of consideration is given to the existence and construction of CFPA. The union, intersection and complement of CFPA can be computed efficiently, and their behavior in decision problems is similar to that of SDPA. Moreover, every complementable PA can be transformed into complementable normal form. Appendix A touches on minimization of PA, while not contributing greatly to the other ideas in this thesis.

Chapter 2

Preliminaries

2.1 Fundamentals

2.1.1 Formal Languages

This chapter will cover the prerequisite knowledge needed to comprehend parametrized automata, beginning with a bare-bones introduction to formal languages. If the reader is already familiar with formal languages, this section can be skipped entirely. If the reader wants to get even better acquainted with formal languages, a more exhaustive introduction can be found in [11, chapter 1.5].

Definition 2.1 (alphabets, words, [11, section 1.5.1 and 1.5.2])**.** In the context of computer science, an alphabet is a non-empty set that may contain a finite or infinite number of elements. The elements of the set are called letters. A word *(over an alphabet D)* is any string or finite sequence of letters (from D). The *length* of a word refers to its number of letters. A word can have a length of 0: The empty word is often denoted ε.

Example 2.2.

1. $D_1 = \{a, b, c\}$ is an alphabet. Examples of words over this alphabet are aaa, ε and $abacbacb$.

2. $D_2 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is an alphabet. Examples of words over this alphabet are 1, 42 or 100000. In the context of formal languages, these words do not carry any mathematical meaning: They *look like* numbers, because humans tend to use these symbols to *represent* numbers.

3. $\mathbb{Z}$, the set of integers, is an infinite alphabet. Some *letters* of this alphabet are 42, 100000 or -5. Examples of words over this alphabet are 642220, 642220 or -100. Since the definition of alphabets as non-empty sets permits letters to be strings of symbols themselves, the notation used so far has been highly ambiguous. The words 642220 (consisting of the letters 64, 222 and 0) and 642220 (consisting of the letters 64 and 2220) are distinct, but there is no way to tell. For situations like this we can introduce an alternative notation: Any word $w = w_1 w_2 \dots w_k$ can also be denoted as $w = (w_1, w_2, \dots, w_k)$. The above words can then be written $(64, 222, 0)$, $(64, 2220)$ and $(-1, 0, 0)$, resolving the ambiguity.

Definition 2.3 (formal languages, [11, section 1.5.3])**.** For an alphabet D, let D^* be the set of all words over D. A formal language over D is any subset $L \subseteq D^*$ of words over D.

Technically, the * in D^* is an operator called the Kleene star which will be discussed in a little while. We also call D^* the universal language. Note that the universal language is always defined in relation to a base alphabet D.

Example 2.4.

1. Let D be any alphabet. The sets $\{\varepsilon\}$, D and D^* are formal languages over D.
2. $\{aac, bcbcbb, cccab, \varepsilon\}$ is a formal language over $D_1 = \{a, b, c\}$.
3. A formal language over $D_2 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is $L = \{w \in D_2^* \mid w = 0$ or the first letter of w is not 0 and the last letter of w is $0, 2, 4, 6$ or $8\}$. Intuitively, the set contains all representations of "even numbers".

Since formal languages are nothing but sets, the union, intersection and complement of formal languages can be constructed.

Definition 2.5 (Boolean and some non-Boolean operations on languages, [11, section 3.1.1])**.** Let $L_1, L_2 \subseteq D^*$ be two formal languages. We define

- the union $L_1 \cup L_2 = \{w \in D^* \mid w \in L_1 \vee w \in L_2\}$,
- the intersection $L_1 \cap L_2 = \{w \in D^* \mid w \in L_1 \wedge w \in L_2\}$,
- the complement $L_1^c = \{w \in D^* \mid w \notin L_1\} = D^* \setminus L_1$,
- the concatenation $L_1L_2 = \{uv \in D^* \mid u \in L_1, v \in L_2\}$, where $uv = (u_1, u_2, \ldots, u_n, v_1, v_2, \ldots, v_k)$ is the concatenation of the words $u = (u_1, u_2, \ldots, u_n)$ and $v = (v_1, v_2, \ldots, v_k)$,
- $L_1^k = \{w_1w_2 \ldots w_k \in D^* \mid w_i \in L_1$ for all $i = 1, \ldots, k\}$, which denotes the concatenation of a language with itself exactly k times,
- and the (Kleene) closure $L_1^* = \bigcup_{i=0}^{\infty} L_1^i$.

Example 2.6. Let $D = \{a, b\}$, $L_1 = \{a, aa, aaa\}$ and $L_2 = \{\varepsilon, a, b, aa, ab, ba, bb\}$. Then

- $L_1 \cup L_2 = \{\varepsilon, a, b, aa, ab, ba, bb, aaa\}$,
- $L_1 \cap L_2 = \{a, aa\}$,
- $L_2^c = \{aaa, aab, aba, \ldots, bbb, aaaa, \ldots\}$, containing all words of a length greater than 3,
- $L_1L_2 = \{a, aa, ab, aaa, aab, aba, abb, aaaa, aaab, aaba, aabb, aaaaa, aaaab, aaaba, aaabb\}$,
- $L_1^0 = \{\varepsilon\}$, $L_1^1 = L_1$, $L_1^2 = \{aa, aaa, aaaa, aaaaa, aaaaaa\}$,
- $L_1^* = \{\varepsilon, a, aa, aaa, \ldots\}$.

There is a subclass of formal languages, the so-called *regular languages*, which is of special interest to us. Regular languages can be described using a kind of notation called *regular expressions*.

Definition 2.7 (regular expressions, [11, section 3.1.2])**.** Let D be a finite alphabet. Regular expressions over D can be defined inductively:

- ε is a regular expressions, denoting the language $L(\varepsilon) = \{\varepsilon\}$ (base case).
- $\emptyset$ is a regular expression, denoting the language $L(\emptyset) = \{\}$ (base case).
- Any letter $a \in D$ is a regular expression, denoting the language $L(a) = \{a\}$ (base case).
- If E_1 and E_2 are regular expressions, then $E_1 + E_2$ is a regular expression, denoting the language $L(E_1 + E_2) = L(E_1) \cup L(E_2)$.
- If E_1 and E_2 are regular expressions, then E_1E_2 is a regular expression, denoting the language $L(E_1E_2) = L(E_1)L(E_2)$.
- If E is a regular expression, then E^* is a regular expression, denoting the language $L(E^*) = L(E)^*$.
- Parentheses: If E is a regular expression, then (E) is a regular expression, denoting the language $L((E)) = L(E)$.

The languages defined by regular expressions are called regular languages.

We introduce some additional notation in order to improve the legibility of regular expressions. For a regular expression E, let E^k denote the regular expression $EE \dots E$ (k times, of course), such that $L(E^k) = L(E)^k$. Also, we permit the use of variable symbols representing sets of letters or words. Careful: If the represented set is not a regular language, the resulting regular expression may also not correspond to a regular language.

Example 2.8. Consider once more $D_2 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and the language $L = \{w \in D_2^* \mid w = 0 \text{ or the first letter of } w \text{ is not } 0 \text{ and the last letter of } w \text{ is } 0, 2, 4, 6 \text{ or } 8\}$. We now have the tools to denote L with a regular expression. A regular expression representing L could be $0 + 2 + 4 + 6 + 8 + (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9)D_2^*(0 + 2 + 4 + 6 + 8)$. In this example, we have used D_2 as a variable symbol to represent the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. L is therefore a regular language.

Although it will not be proven in the scope of this thesis, regular languages are closed under the Boolean operations of intersection, union and complementation [11, section 4.2.1].

2.1.2 First-Order Logic

This section will briefly introduce first-order logic, roughly following the introduction to first-order logic given in [1, section 2.1].

The reader should be familiar with expressions such as "$2 < 1$" or "$5+x = 5 \to x = 0$", using the integers $\mathbb{Z}$. We will keep using this example to explain the different concepts

introduced in this section, and ultimately show how these expressions are defined rigorously in first-order logic.

In order to define logical formulas, we first need to define signatures. A signature clarifies which symbols are used as predicates, functions and variables, and also assigns each function or predicate symbol an arity. Intuitively, the arity defines the number of "input parameters" for each function and predicate.

Definition 2.9 (signature, [1, section 2.1])**.** A signature is a tuple $\Sigma = (\Sigma_p, \Sigma_f, \Sigma_Y, \sigma)$ where

- Σ_p is a set of predicate symbols,
- Σ_f is the set of function symbols,
- Σ_Y is an infinite set of variables,
- and $\sigma : \Sigma_p \cup \Sigma_f \to \mathbb{N}$ is a function mapping the predicate and function symbols to their arity.

The sets Σ_p, Σ_f and Σ_Y are pairwise disjoint.

Some definitions also introduce a set Σ_c of constants. We will instead consider constants to be functions with an arity of 0 (or 0-ary functions).

Example 2.10. The x in "$5 + x$" is a variable; a placeholder for some integer that can be plugged in later. The number of variables is infinite to ensure there are enough of them for arbitrarily long expressions, and for relabeling variables if necessary.

The $+$ in "$5 + x$" is a 2-ary function symbol, and the $<$ in "$2 < 1$" is a 2-ary predicate symbol. The difference between functions and predicates will be explained in a minute. Commonly, functions and predicates use *prefix notation*: A 2-ary function f applied to two inputs a and b is represented $f(a, b)$. Integers use *infix notation*, where the function or predicate symbol is written between its inputs. Therefore, in this example, we skip over the less familiar prefix notation ("$+(5, x)$", "$< (2, 1)$") entirely. Additionally, we use $\approx$ instead of $=$ to denote the equality relation in order to avoid ambiguity.

The symbols 1, 2 and 5 are constants.

A theory of integers (theories will be defined in two minutes) could therefore have the signature $\Sigma^{\mathbb{Z}} = (\Sigma_p^{\mathbb{Z}}, \Sigma_f^{\mathbb{Z}}, \Sigma_Y^{\mathbb{Z}}, \sigma^{\mathbb{Z}})$, where

- $\Sigma_p^{\mathbb{Z}} = \{\approx, <\}$,
- $\Sigma_f^{\mathbb{Z}} = \{\ldots, -2, -1, 0, 1, 2, \ldots, +, -, \cdot\}$,
- $\Sigma_Y^{\mathbb{Z}} = \{x, y, z, \ldots\}$.

The arity of constant functions, such as -1, is $\sigma^{\mathbb{Z}}(-1) = 0$. The arity of all other function and predicate symbols (i. e., $\approx$, $<$, $+$, $-$ and $\cdot$) is $\sigma^{\mathbb{Z}}(\approx) = 2$.

Now that predicate and function symbols have been established, we can define how to combine those into terms and formulae:

Definition 2.11 (terms, formulae, [1, section 2.1])**.** Terms can be defined inductively as either:

- a variable,
- or an n-ary function taking n terms as arguments.

A formula is defined as one of the following:

- an n-ary predicate applied to n terms, where 0-ary predicates represent propositional variables,
- the application of a logical connective $\neg$, $\wedge$, $\vee$, $\rightarrow$ or $\leftrightarrow$ to a formula or formulae,
- the truth symbols $\top$ and $\bot$,
- or the application of a quantifier $\forall$ or $\exists$ to a variable and a formula (we don't really need to care about quantifiers in this context).

Example 2.12. In the context of integers, a term is therefore "something that represents an integer" and a formula is "something that can be true or false". Applying a function to a term or terms results in another term, while applying a predicate to a term or terms results in a formula.

Consider the expression "$5 + x \approx 5 \rightarrow x \approx 0$". It is a formula, constructed by applying the logical connective "$\rightarrow$" to the formulae "$5 + x \approx 5$" and "$x \approx 0$". The formula "$5 + x \approx 5$" is constructed by applying the 2-ary predicate "$\approx$" to the terms "$5 + x$" and "5". The term "$5 + x$" is constructed by applying the 2-ary function "$+$" to the terms "5" and "x".

All "formulae that occur in the construction of a formula" are called subformulae. More rigorously:

Definition 2.13 (subformulae, [1, section 2.1])**.** Let φ and ψ be formulae.

- The only subformula of $\top$, $\bot$ or $p(t_1, \ldots, t_n)$, an n-ary predicate applied to n terms, is the formula itself.
- The subformulae of $\neg\varphi$ are $\neg\varphi$ and the subformulae of φ.
- The subformulae of $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$ or $\varphi \leftrightarrow \psi$ are the formula itself, the subformulae of φ and the subformulae of ψ.

We denote $\varphi \equiv \psi$ if the formulae φ and ψ are identical.

This concludes the discussion of the syntax of first-order logic. We are now capable of constructing well-formed formulae. Next, we turn to the semantics, or interpretation of these formulae.

Definition 2.14 (structure, interpretation, variable assignment, [1, section 2.2])**.** Let there be a signature $\Sigma = (\Sigma_p, \Sigma_f, \Sigma_Y, \sigma)$. A structure $M = (D, I)$ is a tuple consisting of a nonempty set (also called domain) D and an interpretation I. The elements of D are called values. The interpretation I maps each predicate p in Σ_p to a relation $I(p) \subseteq D^{\sigma(p)}$ and each function f in Σ_f to a function $I(f) : D^{\sigma(f)} \rightarrow D$.

A variable assignment is a function $\mu : \Sigma_Y \rightarrow D$ that maps the variables to concrete values in D. The variable assignment is independent from the interpretation I.

Example 2.15. Continuing the integer example, let $\Sigma^{\mathbb{Z}}$ be the signature from example 2.10. For the domain, we naturally choose $\mathbb{Z}$. The interpretation of the predicate "$<$", for example, maps the symbol "$<$" to the $<$-relation as we know it: $I(<) = \{(a, b) \in \mathbb{Z} \mid a < b\}$. The relation $I(<)$ contains all tuples (a, b) for which we wish $a < b$ to hold, such as $(1, 4)$ but not $(2, 2)$ or $(0, -5)$.

The interpretation of a function symbol, for example $+$, is a function $I(+) : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ that maps two integers to their sum (e. g., $I(+)(1, 3) = 4$). We consider constants, such as 0, to be 0-ary functions: $I(0)$ is a function that does not take any input parameters and maps to $0 \in \mathbb{Z}$.

Now, all the previously defined symbols have been assigned a meaning. The final step is to define the evaluation function, which determines whether a formula is true or false.

Definition 2.16 (evaluation of terms and formulae, [1, section 2.1])**.** Let $M = (D, I)$ be a structure and $\mu : \Sigma_Y \to D$ be a variable assignment. The evaluation function $v_{M,\mu}$ evaluates terms and formulae in the following manner:

- Variables are mapped according to the variable assignment: For a variable x, $v_{M,\mu}(x) = \mu(x)$.
- Terms are mapped according to the interpretation function: $v_{M,\mu}(f(t_1, \ldots, t_n)) = I(f)(v_{M,\mu}(t_1), \ldots, v_{M,\mu}(t_n))$ for a term $f(t_1, \ldots, t_n)$ consisting of a function f and n terms $t_1, \ldots, t_n$.
- $v_{M,\mu}(\top) = \mathit{true}$, $v_{M,\mu}(\bot) = \mathit{false}$.
- The evaluation of a formula $p(t_1, \ldots, t_n)$ consisting of an n-ary predicate p and n terms $t_1, \ldots, t_n$ is also defined by the interpretation function: The relation $I(p)$ contains exactly the tuples of values for which we wish $p(t_1, \ldots, t_n)$ to be evaluated to *true*. Therefore, $v_{M,\mu}(p(t_1, \ldots, t_n)) = \mathit{true}$ if and only if $(v_{M,\mu}(t_1), \ldots, v_{M,\mu}(t_n)) \in I(p)$ (*false* otherwise).
- For formulae φ and ψ,
  - $v_{M,\mu}(\neg\varphi) = \mathit{true}$ if and only if $v_{M,\mu}(\varphi) = \mathit{false}$,
  - $v_{M,\mu}(\varphi \vee \psi) = \mathit{true}$ if and only if $v_{M,\mu}(\varphi) = \mathit{true}$ or $v_{M,\mu}(\psi) = \mathit{true}$,
  - $v_{M,\mu}(\varphi \wedge \psi) = \mathit{true}$ if and only if $v_{M,\mu}(\varphi) = \mathit{true}$ and $v_{M,\mu}(\psi) = \mathit{true}$,
  - $v_{M,\mu}(\varphi \to \psi) = \mathit{true}$ if and only if $v_{M,\mu}(\varphi) = \mathit{false}$ or $v_{M,\mu}(\psi) = \mathit{true}$,
  - $v_{M,\mu}(\varphi \leftrightarrow \psi) = \mathit{true}$ if and only if either $v_{M,\mu}(\varphi) = \mathit{true}$ and $v_{M,\mu}(\psi) = \mathit{true}$ hold, or $v_{M,\mu}(\varphi) = \mathit{false}$ and $v_{M,\mu}(\psi) = \mathit{false}$ hold.

Example 2.17. Consider the formula $4 + x < 0$ and a variable assignment $\mu : \Sigma_Y^{\mathbb{Z}} \to \mathbb{Z}$ that maps x to $-5 \in \mathbb{Z}$. Then

- $v_{M,\mu}(x) = -5$, $v_{M,\mu}(4) = 4$, $v_{M,\mu}(0) = 0$,
- $v_{M,\mu}(4 + x) = I(+)(v_{M,\mu}(4), v_{M,\mu}(x)) = I(+)(4, -5) = -1$, and
- $v_{M,\mu}(4 + x < 0) = \mathit{true}$ because $(v_{M,\mu}(4 + x), v_{M,\mu}(0)) = (-1, 0) \in I(<)$.

At this point, the integer example is de facto "complete". We have defined a signature and explained how to build syntactically correct formulae. Then, the introduction of a structure and evaluation function allowed us to assign the formulae a meaning.

If wishing to reason about data structures or algebras using first-order logic, these structures are usually formalized using first-order theories.

Definition 2.18 (theory, [1, section 3.1])**.** A theory $T = (\Sigma, S)$ consists of a signature Σ and a class of structures S. Alternatively, a theory can be defined by a signature Σ alongside a set of axioms.

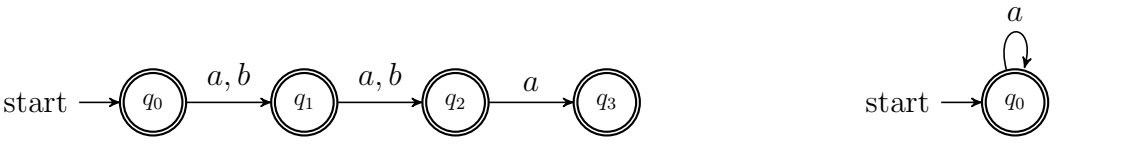
Finally, we introduce the language needed to express the properties of a formula in the context of a theory.

Definition 2.19 (satisfiability, [1, section 3.1])**.** Let $T = (\Sigma, S)$ be a theory. A formula φ is

- *satisfiable in the theory* T, or T*-satisfiable*, if there is a structure $M \in S$ and a variable assignment μ such that $v_{M,\mu}(\varphi) = \mathit{true}$ (the formula can evaluate to *true*),
- T*-valid* if $v_{M,\mu}(\varphi) = \mathit{true}$ holds for *every* structure $M \in S$ and variable assignment μ (the formula will always evaluate to *true*),
- T*-unsatisfiable* if it is not T-satisfiable (the formula will always evaluate to *false*),
- *counter-*T*-satisfiable* or T*-invalid* if it is not T-valid (the formula can evaluate to *false*),
- or T*-contingent* if it is both T-satisfiable and counter-T-satisfiable (the formula can evaluate to both *true* and *false*).

A theory is *decidable* if there exists an algorithm that terminates for every T-formula with *yes* if the formula is T-valid and *no* if it is not.

2.2 Finite-State Automata



(a) An automaton for $L_1 \cup L_2$. (b) An automaton for L_1^*.

Figure 2.1: Examples of finite-state automata representing languages from example 2.6.

We have previously encountered regular expressions as a way to define regular languages. Another common device to identify and define regular languages are *finite-state automata* ([11, chapter 3.2]). For a visual approach, finite-state automata can be pictured as directed graphs in which the vertices are called states and the directed edges transitions. One state is marked as the initial state, and some states are marked as "accepting". The transitions are labeled with letters from the finite alphabet. This way, each path from the

initial state to an accepting state spells out a word when noting, in order, the letters that label the traversed transitions. The language of an automaton is defined as the set of all words that can be constructed in such a manner. In particular, this language is regular. As such, each finite-state automaton corresponds to exactly one regular language, which is the set of words that define paths leading to accepting states.

This informal, intuitive description will now be formalized:

Definition 2.20 (finite-state automata)**.** A finite-state automaton (FSA) is a tuple $A = (\Sigma, Q, q_0, \delta, F)$ consisting of

- a finite alphabet Σ,
- a finite set of states Q,
- an initial state $q_0 \in Q$,
- a transition relation $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$, where $\varepsilon \notin \Sigma$ represents the empty word, and
- a finite set $F \subseteq Q$ of accepting states.

The word w is accepted by the FSA A if it has an accepting run in A. The set of all words accepted by A is a formal language, denoted $L(A)$.

The definition of a run will follow shortly. First, we should think a little more about automata, and try to shift our understanding towards automata as acceptors of regular languages.

An FSA can be thought of as a machine that corresponds to exactly one language. The FSA takes a word as input, and then either rejects or accepts the word as part of its language. The machine's decision making process can be divided into steps. In each step, the leftmost letter from the input word is consumed. The automaton then enters a new state (which may coincide with the previous state), dictated by the letter, the current state and the transition relation. If the automaton is in an accepting state after consuming the last letter of a word, the word is accepted.

In some steps, the current letter and state permit more than one successor state. Such automata are called non-deterministic, and in line with definition 2.20, a word is accepted if at least one possible sequence of transitions terminates in an accepting state. It does not matter if different choices of transitions lead to a termination in a non-accepting state.

Likewise, there may be combinations of letters and states such that no successor state exists. In this case, the automaton cannot terminate in an accepting state because the input word has not been consumed entirely.

All of these notions will now be formalized in the following definition.

Definition 2.21. A run (or complete run) of a word $w \in \Sigma^*$ in an FSA A is a sequence of states $q_0, q_1, \ldots, q_n$ and transitions $(q_0, w_1, q_1), (q_1, w_2, q_2), \ldots, (q_{n-1}, w_n, q_n)$ such that q_0 is the initial state and, when removing all w_i where $w_i = \varepsilon$, $w_1 w_2 \ldots w_n = w$ (sometimes, we also call this the path of w through A). A run is accepting if its final state q_n is an accepting state. The *length* of the run is n.

This definition permits some of the letters w_i to correspond to ε. A transition $(q_{i-1}, \varepsilon, q_i)$ is called an ε-transitions. No letter of the input word is consumed when using an ε-transition.

Let $w_1 w_2 \dots w_k$ be a strict prefix of w, meaning that there is a nonempty word $u_1 u_2 \dots u_m \in \Sigma^*$ such that $w_1 w_2 \dots w_k u_1 u_2 \dots u_m = w$. Then a complete run of $w_1 \dots w_k$ is called an incomplete run of w.

Now let $w = w_1 \dots w_n$ be a word, let $(q_0, w_1, q_1), (q_1, w_2, q_2), \dots, (q_{k-1}, w_k, q_k)$ be a complete run of the strict prefix $w_1 \dots w_k$ and let $w_{k+1} \in \Sigma$ be the "next" letter of w, such that $w_1 \dots w_k w_{k+1}$ is a prefix of w. Let q_{k+1} be a state. A transition (q_k, u, q_{k+1}) exiting the current state q_k is viable if $u = \varepsilon$ or $u = w_{k+1}$. If no exiting transition is viable, we say that the run dies at state q_k.

FSA originated in the 1940's ([19]), and are thus well-researched. Analogue to regular languages, FSA are closed under Boolean operations: Given two FSA A and B, it is possible to construct FSA accepting the union $L(A) \cup L(B)$, intersection $L(A) \cap L(B)$, and complement $L(A)^c = \Sigma^* \setminus L(A)$. We will only touch on the complementation problem in more detail, as it will prime us for the challenges tied to complementation of other types of automata.

In relation to complementation of FSA, determinism is an important concept: In a deterministic FSA, every word completes exactly one run. Deterministic FSA are easy to complement, because every word which completes a run in a non-accepting state has to be part of the complement.

Definition 2.22 (determinism of finite-state automata [11, section 2.2.1])**.** A FSA is deterministic if there is exactly one possible complete run for each word $w \in \Sigma^*$. This criterion is met if and only if for every letter $a \in \Sigma$ and every state $q \in Q$, there is exactly one state $p \in Q$ such that $(q, a, p) \in \delta$.

Not every FSA is deterministic, but every FSA can be transformed into an equivalent FSA that is deterministic.

Theorem 2.23 (determinization of FSA)**.** For every FSA A, there is an equivalent FSA A' that is deterministic. Two automata A and A' are equivalent if $L(A) = L(A')$.

Proof. A detailed version of this proof can be found in [11, section 2.3.5].

An FSA can be determinized using a subset construction: A new automaton is created, whose states correspond to subsets of states of the original automaton. A transition (S, a, T) between two set-states S and T exists if T consists of all states that can be reached from states in S via a sequence of transitions containing exactly one a-transition and an arbitrary number of ε-transitions.

Starting with an FSA $A = (\Sigma, Q, q_0, \delta, F)$, let $P(Q)$ be the power set of Q. Let $\delta_d \subseteq P(Q) \times \Sigma \times P(Q)$ consist of all tuples (S, a, T) such that $T = \{p \in Q : (q, a, p) \in \delta$ for some $q \in S\}$. Let $q_d = \{q_0\}$, and let $F_d = \{S \in P(Q) : S \cap F \neq \emptyset\}$ be the set of all subsets of Q that contain at least one accepting state.

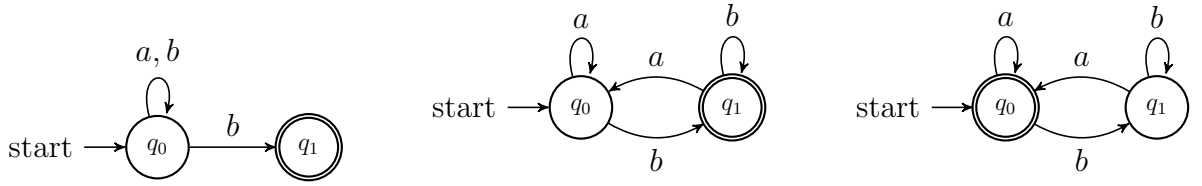
The resulting FSA $A' = (\Sigma, P(Q), q_d, \delta_d, F_d)$ is obviously deterministic and equivalent to A. The latter can be shown by an induction proof. □

Combining the previous observations, an algorithm for complementing arbitrary PA can be constructed.

Theorem 2.24 (complementation of FSA [11, section 4.2.1])**.** Finite-state automata are closed under complementation. Given an FSA A, an FSA A^c such that $L(A^c) = L(A)^c$ always exists and can be constructed. We say that A^c is the complement automaton of A.

Proof. Let $A' = (\Sigma, Q, q_0, \delta, F)$ be a deterministic FSA equivalent to A, which exists according to theorem 2.23. Then the automaton $A^c = (\Sigma, Q, q_0, \delta, Q \setminus F)$, obtained by "swapping" all states of A' such that the accepting states become non-accepting and vice versa, identifies the complement of A' and therefore of A.

Since each word w only completes one run through A', $w \in L(A')^c$ if and only if its unique run terminates in a non-accepting state of A'. Similarly, if the run of w terminates in an accepting state of A', it cannot be part of the complement of $L(A')$. □



(a) An automaton for L_1.

(b) A deterministic automaton for L_1.

(c) An automaton for the complement of L_1.

Figure 2.2: Examples of finite-state automata, where the language L_1 contains all words ending in the letter b.

Example 2.25. Let $\Sigma = \{a, b\}$ and $L_1 = \{w \in \Sigma^* : \text{the last letter of } w \text{ is } b\}$. Figure 2.2 displays non-deterministic and deterministic FSA identifying L_1, and an FSA identifying L_1^c.

Finally, we give a short introduction to a number of classical *decision problems* in automata theory.

Definition 2.26 (decision problems, [21, chapter 3.1, definition 3.6], [1, section 2.6.2])**.** In computer science, a decision problem is a yes-or-no-question. It is commonly represented as a base set S and a subset $P \subseteq S$ of positive cases. A decision problem is decidable if there exists an algorithm that halts on any input $s \in S$ and returns "yes" if and only if $s \in P$, and "no" otherwise. A decision problem is semidecidable if there exists an algorithm that takes any element $s \in S$ as input and returns "yes" if and only if $s \in P$, and does not terminate otherwise.

Example 2.27. Some decision problems are of special interest in automata theory. To investigate the expressiveness of an automata class, such as deterministic finite automata or non-deterministic finite automata, we often consider their "difficulty" regarding the following decision problems:

- *Universality:* The universality problem is the problem of deciding whether an automaton's language is universal. The problem is decidable if there exists an algorithm that takes any automaton A over an alphabet D as input, always terminates and returns "yes" if $L(A) = D^*$ and "no" if there exists a word that does not complete an accepting run in A.

- *Non-Emptiness:* The non-emptiness problem is the problem of deciding whether the language of an automaton A is not the empty language, i. e. whether there exists some $w \in D^*$ such that $w \in L(A)$.

- *Equivalence:* The equivalence problem is the problem of deciding whether two automata A and B identify the same language, $L(A) = L(B)$.

- *Containment:* The containment problem is the problem of deciding whether the language of an automaton A is a subset of the language of an automaton B, $L(A) \subseteq L(B)$.

- *Membership:* The membership problem is the problem of deciding whether a word w is accepted by an automaton A. An algorithm deciding the membership problem therefore takes an automaton A and a word w as input, and returns "yes" if $w \in L(A)$.

- *Reachability:* The reachability problem is the problem of deciding whether a state of an automaton is reachable. An algorithm deciding the reachability problem takes an automaton A and a state q of A as input, and returns "yes" if there exists a word that has a run terminating in q.

When a problem is said to be "decidable for deterministic FSA", this refers to deterministic FSA being the base set of the problem. An algorithm deciding the problem can therefore assume all input FSA to be deterministic.

A related question, which is also of interest to us, refers to a problem's computational complexity (see [1, section 2.6.3]. If a problem is decidable, how quickly can it be solved by an algorithm? The complexity classes most important to us are P (solvable in polynomial time), NP (solvable non-deterministically in polynomial time) and PSPACE (solvable in polynomial space).

Theorem 2.28. The following statements hold:

- The non-emptiness, reachability, and membership problems are in P for both deterministic and non-deterministic FSA ([11, chapter 4.3]).

- The universality, equivalence and containment problems are in P for deterministic FSA (since non-emptiness and reachability are in P), and PSPACE-complete for non-deterministic FSA ([16]).

2.3 Automata for Infinite Alphabets

FSA, as introduced before, are by construction only capable of identifying finite words based on finite alphabets. In order to extend this model to finite words from infinite alphabets, the scientific community has come up with a multitude of modifications to FSA. We will describe three of these classes of automata in more detail.

The first will be *register automata* which, as one of the oldest attempts to extend finite-state automata to infinite alphabets, have an enormous historical significance. If FSA are a distant ancestor of parametrized automata, register automata are their uncle. We will also meet the parents: *Variable automata* and *symbolic automata*, two very different classes of automata which are both extended by parametrized automata.

2.3.1 Register Automata

Initially introduced as "finite memory automata" (see [14]), register automata are one of the oldest models designed to handle languages over infinite alphabets, and nowadays exist in plenty of variations. One such variation, *symbolic register automata*, will be briefly described at the end of the chapter. For now, we will stick to the original version.

Definition 2.29 (register automata, [14, chapter 2, definition 1])**.** Let D be an infinite alphabet. A register automaton (RA) is a tuple $A = (Q, q_0, u, \rho, \delta, F)$ of

- a finite, non-empty set of states Q,
- an initial state q_0,
- an initial assignment $u = w_1^0 w_2^0 \dots w_r^0 \in (D \cup \{c\})^r$, where c is a letter not occurring in Σ and r is the "storage size",
- a (partial) reassignment function $\rho : Q \to \{1, 2, \dots, r\}$,
- a transition relation $\delta \subseteq Q \times \{1, 2, \dots, r\} \times Q$ and
- a set $F \subseteq Q$ of final states.

The transitions of register automata are not labeled with letters, but with positions in the register. Starting at a state q with a current letter a, a transition $(q, k, p) \in \delta$ is viable in either of these cases:

- The current letter a corresponds to the k-th entry of the register, $a = w_k$. If chosing the transition, no adjustments need to be made to the register.
- The current letter a does not equal any other entry in the register, and $\rho(q) = k$. If the transition is chosen, the register entry w_k has to be overwritten with a.

In either case, the k-th entry of the register will store the letter a after the transition. A word is accepted by the register automaton if it completes a run in this manner in an accepting state.

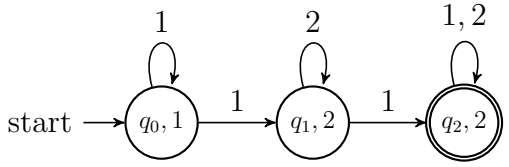
Since the reassignment function ρ is static, we can improve the comprehensibility of the depictions of register automata below by adding the label $\rho(q)$ to each state q.

Example 2.30. Let $R = (\{q_0, q_1, q_2\}, q_0, (c, c), \rho, \delta, \{q_2\})$ be the automaton in figure 2.3a, where D is an infinite alphabet not containing the letter c from the initial assignment, and $\rho(q_0) = 1, \rho(q_1) = \rho(q_2) = 2$. Two transitions start in the initial state, both of which compare the upcoming letters to the first entry of the register, $u(1)$. Since the second entry of the register, $u(2)$, is still assigned c, every upcoming letter can be read and then stored in the first entry. The choice to stay in q_0, or else take the transition leading to state q_1, is non-deterministic.

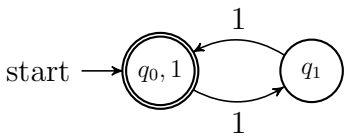
If state q_1 is entered upon some letter a, the letter a will be stored in the first entry of the register indefinitely, since $\rho(q_1) = 2$ and upcoming letters will be stored in the second entry of the register. Now, if an upcoming letter does not equal a, it will be stored in the second entry of the register and the run remains in state q_1. If an upcoming letter equals

a, the transition leading to the accepting state q_2 has to be taken. In q_2, both the letter a and any letter that does not equal a will cause the run to loop back to q_2.

The automaton accepts a word if some letter appears twice and if the run happens to leave state q_0 while said letter is stored. Runs of words that do not contain any repeat letters will not be able to exit state q_1.



(a) A register automaton identifying all words in which some letter appears at least twice.



(b) A register automaton identifying words in which each letter appears in a pair.

Figure 2.3: Examples of RA.

Finite-state automata can always be complemented. Register automata have lost this property: For a register automaton A, a corresponding register automaton A^c such that $L(A^c) = L(A)^c$ may not exist.

Proposition 2.31. RA are not closed under complementation.

Proof. Consider the register automaton R from example 2.30. A register automaton identifying the complement of $L(R)$ would have to identify exactly the words in which no letter repeats. As shown in [14, section 2, proposition 5], such a register automaton does not exist. □

Register automata are capable of identifying a wide range of languages, which leads to a recurring problem in automata theory: The more powerful the class of automaton, the more does the complexity of operations increase and the more likely it is that important decision problems are undecidable. For the same reason, the ability to complement RA has been lost. The universality problem (and therefore, also the equivalence and containment problems) is undecidable for register automata (see [18]), and the membership and non-emptiness problems are NP-complete (see [20]).

2.3.2 Variable Automata

An alternative to register automata are variable automata, which are also capable of comparing input letters to a set of stored parameters (here called bounded variables). Unlike register automata, the parameters of variable automata cannot be reassigned. The content of this subsection largely follows [10].

Definition 2.32 (variable automata, [10, section 2])**.** A variable finite automaton (VA) $A = \langle D, M\rangle$ consists of an infinite alphabet D and an FSA $M = (\Sigma, Q, q_0, \delta, F)$, called the *pattern automaton* of A. The alphabet of M, which we call $\Sigma = C \cup Y \cup \{z\}$, consists of:

- a finite set of constant letters $C \subseteq_{fin} D$,

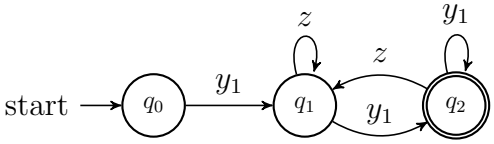
- a finite set of bounded variables $Y = \{y_1, \ldots, y_n\}$,
- and a free variable z.

The sets C, Y and $\{z\}$ are pairwise disjoint.

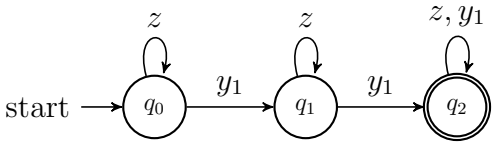
Intuitively, for each run of a word through the automaton, the bounded variables are assigned letters from D according to some assignment function $\mu : Y \to D$. No two bounded variables may be assigned the same letter, and no bounded variable may be assigned a constant letter. A transition (q, y_i, p) is viable if the word's current letter matches the assignment to y_i, and a transition (q, z, p) is viable if the current letter does not match the assignment to any bounded variable. A word is accepted if there exists a variable assignment such that the word completes a run in an accepting state.

Formally, a word $w = (w_1, \ldots, w_n) \in D^*$ is accepted by A if there is a word $v = (v_1, \ldots, v_n) \in L(M) \subseteq \Sigma^*$ and an injective variable assignment $\mu : Y \to D \setminus C$ such that $\mu^{-1}(w) = v$, where $\mu^{-1}(w_i) = \begin{cases} w_i & w_i \in C \\ y_k & y_k \in Y \text{ and } \mu(y_k) = w_i \\ z & \text{else.} \end{cases}$

In the literature, v is called a witnessing pattern of w.



(a) The pattern automaton of a deterministic VA V_1 that accepts all words whose first and last letter match.



(b) The pattern automaton of a VA V_2 accepting all words that repeat some letter.

Figure 2.4: Examples of VA.

Similar to register automata, we can observe:

Proposition 2.33 ([10, section 3, theorem 3])**.** Variable automata are not closed under complementation.

Proof. As a counterexample, we use the same language that has been used in the corresponding proof for register automata. The language $L = \{w = w_1 w_2 \ldots w_n, w_i = w_j \text{ for some } i, j \leq n, i \neq j\}$ can be expressed via a VA, as seen in figure 2.4b. However, there is no VA corresponding to the complement of the language.

Assume that a VA identifying the complement exists, using n bounded variables. Let w be a word consisting of $n+2$ non-constant, distinct letters. Since there are more letters in w than bounded variables, w's run through the automaton has to traverse a transition labeled z at least twice.

Let w_i and w_j be two letters of w (in the positions $i \neq j$) that are using the z-transitions. Let $v \in D^*$ match w in all positions except j, where $v_j = w_i$. Then $v \in L$, as the letter $v_i = v_j$ occurs twice. However, v's run traverses the assumed complement automaton on the same path as the run of w, and therefore ends in an accepting state. Therefore, an automaton accepting the complement of L cannot exist. □

The VA V_2 depicted in figure 2.4b helps illustrate how the assignment of variables forces the trajectory of a run. Consider the word $w = (1, 3, 3)$. The word only terminates in the accepting state q_2 if $\mu(y_1) = 3$. When using the assignment $\mu(y_1) = 1$, the run terminates in state q_1 instead. Although the pattern automaton of V_2 is deterministic, there are words that can complete more than one distinct run in V_2.

The VA V_1 in figure 2.4a is deterministic in the sense that each word only completes one distinct run. Consider $w = (2, 2, 2)$. Only the assignment $\mu(y_1) = 2$ will permit the run to exit state q_0. Any other variable assignment will cause the run to die, as there is no viable transition exiting q_0. After this first step, each transition taken by the word's run is predetermined.

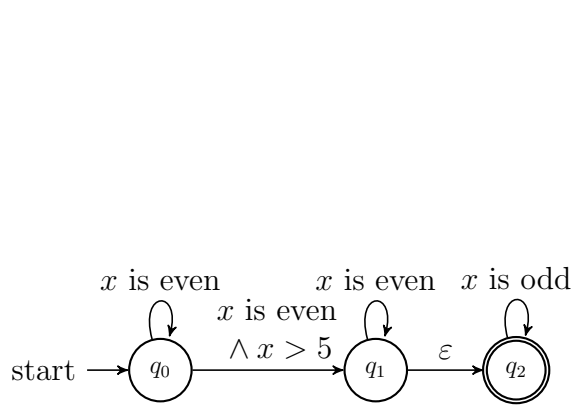
When working with VA, it is therefore important to distinguish between different notions of determinism. Parametrized automata will inherit this trait.

Example 2.34. The languages identified by VA and RA, respectively, are incomparable. For example, the language $L_R = \{w = (w_1, \ldots, w_n) \in D^* \mid w_{2i} = w_{2i-1} \forall 1 < 2i \leq n$ and n is even$\}$ can be identified by a RA, but not by a VA. The language $L_A = \{w = (w_1, \ldots, w_n) \in D^* \mid w_i \neq w_n \forall i < n\}$ containing all words whose last letter is different from all previous letters, on the other hand, can be identified by a VA but not by a RA, as RA cannot compare for inequality.

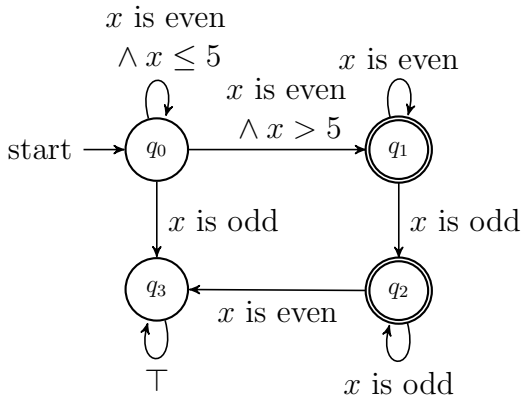
For VA, the non-emptiness problem is NL-complete and the membership problem is NP-complete, while universality and containment are undecidable (all [10, section 4]).

2.3.3 Symbolic Automata

Both register automata and variable automata can store and compare letters of a word. Symbolic automata (SFA, [7]) are using a very different approach to handle infinite alphabets: Their transitions are labeled with logical formulae, also called guards. A transition is viable if the current letter satisfies the conditions posed in the corresponding formula, or guard. Since SFA cannot compare letters of a word and VA cannot apply logical formulae to letters, both kinds of automaton are incomparable and are suited for different kinds of languages.



(a) An SFA that is non-deterministic and uses ε-transitions



(b) A deterministic version of the same SFA.

Figure 2.5: Examples of SFA.

In this thesis, all automata are assumed to use the same theory to ensure compatibility. Examples will often use the theory of real numbers.

Definition 2.35 (symbolic finite-state automata, [7, section 2, definition 1])**.** A symbolic finite automaton is a tuple $A = (M, Q, q_0, \delta, F)$ of

- a structure $M = (D, I)$, where D is an infinite alphabet,
- a finite set of states Q,
- an initial state $q_0 \in Q$,
- a transition relation $\delta \subseteq_{fin} Q \times (\Phi \cup \{\varepsilon\}) \times Q$ and
- a finite set $F \subseteq Q$ of accepting states.

Φ denotes the set of all formulae using only one free variable x. Transitions labeled with ε, which shall be a symbol not occurring in Φ, refer to ε-transitions analogously to finite-state automata. A word $w \in D^*$ is accepted if there exists a sequence of letters $a_1, \ldots, a_n \in D \cup \{\varepsilon\}$, a family of variable assignments $(\mu_{a_i})_{i=1,\ldots,n}$, and a complete run $(q_{i-1}, \varphi_i, q_i)$ for $i = 1, \ldots, n$ in A such that

- $a_1 a_2 \ldots a_n = w$,
- $\mu_{a_i}(x) = a_i$ for all $i = 1, \ldots, n$ where $a_i \neq \varepsilon$, i. e., x is assigned the value of the current letter in every step,
- $\varphi_i = \varepsilon \Leftrightarrow a_i = \varepsilon$,
- $v_{M,\mu_{a_i}}(\varphi_i) = \mathit{true}$ for $i = 1, \ldots, n$ where $a_i \neq \varepsilon$, i. e., all involved formulae are true, and
- $q_n \in F$, i. e., the path ends in an accepting state.

SFA and FSA behave very similarly, because based on the guard formulae, an SFA's infinite alphabet can be partitioned into a finite number of equivalence classes whose members have identical properties within the automaton. An SFA is *deterministic* if in each state and for each letter, exactly one exiting transition is viable. Algorithms for determinizing SFA (including elimination of ε-transitions) are well-known (for further reading, see [22, section III]). A deterministic SFA can be complemented easily by changing the set of accepting states from F to $Q \setminus F$.

Since every symbolic automaton can be determinized, symbolic automata are closed under complementation.

2.3.4 Other Classes of Automata

There are other classes of automata that share traits with parametrized automata and should be mentioned for the purpose of disambiguation.

Symbolic register automata ([4]) combine register and symbolic automata. Different from parametrized automata, the symbolic and register components run "in parallel"

without interacting: Each transition is labeled with a pair of a logical formula and register instructions that are both applied to the input letter independently. In other words, the letters previously stored in registers cannot occur in logical formulas.

Similar to parametrized automata, symbolic register automata are closed under union and intersection, but not under complementation unless the automaton is deterministic. The non-emptiness problem is decidable, but decidability of the containment and equivalence problems are only shown for deterministic symbolic register automata.

Parametric semilinear data automata ([9]) implement parametric semilinear data logic, an extension of linear temporal logic that permits, for example, letter counting. Parametrized automata are not restricted to a single logic; the complexity of decision problems such as the non-emptiness problem depends on the underlying theory. The non-emptiness problem for parametric semilinear data automata is in NEXP.

Extended symbolic finite automata ([6]) are symbolic automata capable of reading multiple adjacent input letters in one transition. Extended symbolic finite automata are closed under union, but not under intersection or complement, and the non-emptiness problem is decidable while the universality and equivalence problems are not.

Chapter 3

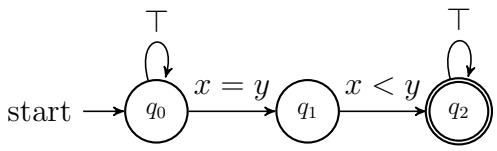
Parametrized Automata

3.1 Definition and Notations

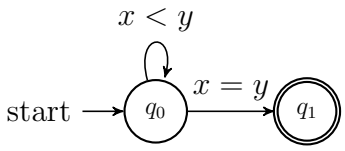
After the groundwork of chapter 2, we can finally properly introduce parametrized automata (PA), the star of this thesis, and start with original research. PA are combining traits of both symbolic automata (SFA) and variable automata (VA): Their transitions are labeled using formulae, called guards, and additionally the formulae may include a finite number of assignable variables or parameters. Once the parameters are assigned fixed values, the resulting automaton behaves identically to an SFA. A word is accepted by a PA if there exists an assignment of parameters such that the word is accepted in the resulting SFA.

For this purpose, let $\{x\} \cup Y$ be a set of variables such that $x \notin Y = \{y_1, y_2, \dots\}$ and Φ be the set of formulae using the variables $\{x\} \cup Y$ (Y denoting the infinitely large set of parameters). Similar to the definition of symbolic automata, x will continue to be the placeholder variable for the current letter, and is not considered a parameter.

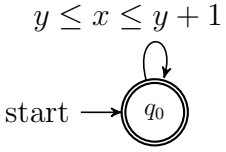
Examples of PA can be seen in figure 3.1.



(a) A_1, identifying unsorted words.



(b) A_2, identifying words whose last letter is largest.



(c) A_3, identifying words whose letters fall within an interval of length 1.

Figure 3.1: Examples of parametrized automata using the theory of real numbers. In all three examples, y denotes the single parameter.

Definition 3.1 (parametrized automata)**.** A parametrized automaton is a tuple $A = (M, Q, q_0, \delta, F)$, where

- $M = (D, I)$ is a structure and D is an infinite alphabet,
- Q is a finite set of states,

- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting states, and
- $\delta \subseteq_{fin} Q \times \Phi \times Q$ denotes the transition relation using a set of formulae Φ.

A word $w = (w_1, \ldots, w_k) \in D^*$ is accepted by the automaton A if there exists a family of variable assignments $(\mu_i)_{i=1,\ldots,n}$ and a sequence of transitions (q_0, φ_1, q_1), (q_1, φ_2, q_2), $\ldots$, $(q_{k-1}, \varphi_k, q_k)$ in A, called a complete run, such that

- $\mu_i(y) = \mu_j(y)$ for all $i, j \in \{1, \ldots, n\}$ and $y \in Y$,
- $\mu_i(x) = w_i$ for all $i = 1, \ldots, n$, i. e., x is assigned the value of the current letter in every step,
- $v_{M,\mu_i}(\varphi_i) = \mathit{true}$ for $i = 1, \ldots, n$, i. e., all involved formulae are true, and
- $q_n \in F$, i. e., the run terminates in an accepting state.

We will always assume that a PA is *normalized*, i. e., for two states $q, p \in Q$ there is at most one transition $(q, \varphi, p) \in \delta$. Therefore, a PA with n states will always have at most n^2 transitions. This does not restrict the power of PA, as a set of transitions $\{(q, \varphi_1, p), \ldots, (q, \varphi_k, p)\}$ connecting q and p can always be replaced by the single transition $(q, \varphi_1 \vee \cdots \vee \varphi_k, p)$. We also assume that the underlying theory T is decidable, i. e., satisfiability of T-formulae is decidable.

Since Y is infinite, each variable assignment μ_i contains a lot of redundant information about the variables that do not occur in formulae of A. Additionally, the placeholder variable x has to be constantly reassigned, conforming to a word's current letter. We will therefore introduce the parameter assignment, which only contains information about the finite number of variables that occur in the formulae and do not represent the current letter. The parameter assignment is constant throughout a run.

Notation 3.2 (parameter assignment)**.** Since δ is finite, the occurring formulae only make use of a finite set of variables, which we will name $Y_A = \{y_1, \ldots, y_k\}$. Due to the condition that $\mu_i(y) = \mu_j(y)$ for all $i, j \in \{1, \ldots, n\}$ and $y \in Y$, there exists a well-defined function $\mu : Y_A \to D$ such that $\mu(y) = \mu_i(y)$ for all $i = 1, \ldots, n$ and $y \in Y_A$. We refer to μ as a parameter assignment (as opposed to a *variable* assignment, which includes the variable x). For any parametrized automaton A using the parameters Y_A, let $\Theta = \{\mu : Y_A \to D\}$ be the set of all possible parameter assignments. We say A has k parameters.

A PA can be "constricted" to one parameter assignment, obtaining a symbolic automaton:

Notation 3.3. Let $A = (M, Q, q_0, \delta, F)$ be a PA using the finite parameter set Y_A and $\mu : Y_A \to D$ be a parameter assignment. We obtain a new automaton A_μ by replacing every variable $y_i \in Y_A$ occurring in a transition formula $\varphi \in \Theta$ with its assigned value $\mu(y_i) \in D$. Since constants now substitute for all variables save x, A_μ is a symbolic automaton accepting exactly the words that are accepted by A using the parameter assignment μ. Clearly, $L(A_\mu) \subseteq L(A)$ and $L(A) = \bigcup_{\mu \in \Theta} L(A_\mu)$.

A_2 in figure 3.1 illustrates how the choice of parameters influences a word's run: A word $w \in L(A_2)$ is only identified correctly if the last letter of w is assigned to y. Only then will the run terminate in the accepting state. If $\mu(y)$ is too large, the run will never transition from q_0 to q_1 and end in a non-accepting state. If $\mu(y)$ is too small, the run will either transition to state q_1 too early and die because there are no exiting transitions, or the run will die in state q_0 as soon as w reaches a letter larger than $\mu(y)$, as no exiting transition is satisfied.

If a word v is not part of $L(A_2)$, it will never terminate in q_1. Therefore, the automaton works correctly.

This thesis is focused on the complementation problem for parametrized automata. Does every PA A have a complement PA A^c such that $L(A)^c = L(A^c)$? And how could this complement PA be computed? Since the former question can be answered using nothing but the meager tools that have just been defined, it will not be delayed further:

Theorem 3.4. PA are not closed under complementation, i. e., there is a PA A such that no PA A^c with the property $L(A^c) = L(A)^c$ exists.

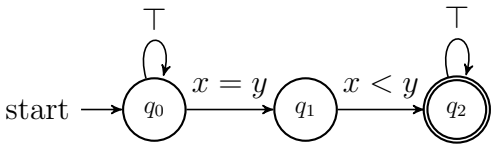


Figure 3.2: A_1 as seen in figure 3.1, a PA that cannot be complemented.

Proof. Consider example A_1 seen in figure 3.2, a PA using the theory of real numbers which randomly assigns a letter w_i of a word $w = (w_1, \ldots, w_k)$ to its parameter y and accepts the word if the succeeding letter is smaller than $w_i = y$. As such, A_1 identifies unsorted words: $L(A_1) = \{w = (w_1, \ldots, w_k) \in D^* \mid \exists 1 \leq i < k : w_i > w_{i+1}\}$. Note that, by this definition, a word is sorted if its letters occur in *ascending* order. An example of a sorted word is $(1, 2, 3)$, while $(1, 3, 2)$ is not a sorted word. The complement of $L(A_1)$ is the set of all sorted words $w = (w_1, \ldots, w_k)$ with the property $i < j \Rightarrow w_i \leq w_j$.

We will prove that a complement automaton identifying $L(A_1)^c$ cannot exist using a proof by contradiction that has a similar flavor as the pumping lemma (see [11, section 4.1]). Assume for contradiction that such an automaton A_1^c exists, having n states. Let $w = (1, 2, 3, \ldots, 2n + 1)$. Since $w \in L(A_1)^c$, there is a parameter assignment μ such that w completes an accepting run in $(A_1)_\mu$. Then by a counting argument, we can argue the run of w in A_1^c traverses some state q thrice, creating a loop of length greater than 1. This loop is traversed by a subword $(i, i + 1, \ldots, i + k)$ of w where $k \geq 1$.

Therefore, the word $w' = (1, 2, \ldots, i + k - 1, i + k, i, i + 1, \ldots, 2n + 1)$ obtained by repeating the found loop completes an accepting run in A_1^c. However, w' is not a sorted word. Therefore, A_1^c does not work directly and a PA identifying all sorted words cannot exist. □

This result is not surprising, given that VA and RA are also not closed under complementation and the models share similar traits. Yet so far, we have not proven in detail how parametrized automata relate to the other classes of automata described in section 2.3. Intuitively, PA subsume variable and symbolic automata, meaning that if a language

can be recognized by a symbolic automaton or a variable automaton, it can also be recognized by a suitable parametrized automaton. We will now prove this intuition formally, and also see that PA and register automata are incomparable.

Theorem 3.5 (automata hierarchy)**.** Parametrized automata subsume variable and symbolic automata, but not register automata.

The statement is visualized in figure 3.3.

Proof.

- *Symbolic automata*: An SFA is a PA in which no parameters occur.
- *Variable automata*: In order to create a PA equivalent to a given VA, it is not sufficient to create a "copy" of the VA where each transition (q, y_i, p) is replaced by a transition $(q, x = y_i, p)$. Compared to PA, VA impose additional restrictions on possible parameter assignments. No two bounded variables can be assigned the same value or a constant, and the free variable z may only correspond to values that are not assigned to a bounded variable. PA do not come with the same restriction, so some inequality checks have to be added manually. Also, PA are usually normalized while VA are not.

  Let $\mathcal{A} = \langle D, V \rangle$ be a VA with pattern automaton $V = (\Sigma, Q, q_0, \delta, F)$. For two states $q, p \in Q$, let $S_{(q,p)}$ be the set of all letters $s \in \Sigma$ such that $(q, s, p) \in \delta$.

  Let $M = (D, I)$ be a structure that permits an equality relation $I(=)$, ensuring that letters in D can be compared for equality and inequality. A PA $A_p = (M, Q, q_0, \delta_p, F)$ that is equivalent to $\mathcal{A}$ shall make use of k parameters $y_1, \ldots, y_k$, named like the bounded variables of V for convenience. All constant symbols from Σ need to be letters from D and therefore can still be treated like constants in A_p. Let the sets Q and F and the state q_0 be equal to the sets and state Q, F and q_0 of V.

  In order to obtain a normalized PA, each transition between the states q and p in A_p needs to represent the set of all transitions between the states q and p in V. All variables and constants which permit transition from q to p in V are contained in the set $S_{(q,p)}$, so a straightforward translation of $\mathcal{A}$ to the language of PA might replace each transition (q, s, p) in V with $(q, x = s, p)$ in P. Normalizing, we obtain a transition (q, φ', p) where $\varphi' \equiv \bigvee_{s \in S_{(q,p)} \setminus \{z\}} x = s$. This leaves the z variable, which represents all letters not assigned to another variable or a constant. If $z \in S_{(q,p)}$, then $\varphi' \equiv \bigwedge_{s \notin S_{(q,p)}} x \neq s$ can be used instead. We are not finished yet, however.

  VA have the additional requirement that each letter of D can be represented by exactly one symbol of Σ: No variable may be assigned a letter that is already assigned to a different variable or a constant, and z may only represent letters that are not already represented by a variable or a constant. A messy way of implementing this requirement in A_p might be to add the necessary inequality checks (e. g., $y_i \neq y_j$ for every pair of variables y_i, y_j) to every single transition. This leads to a quadratic blowup in the length of transition guards, adding roughly $|\Sigma|^2$ subformulae to each guard φ'. Luckily, there is a much more elegant solution.

We can eliminate most of the described inequality constraints by redefining the constraints in relation to x.

Let the finite relation $\delta_p \subseteq_{fin} Q \times \Phi \times Q$ contain the following transitions:

- *Case 1.* If $z \notin S_{(q,p)}$ for $q, p \in Q$: Let $(q, \varphi, p) \in \delta_p$ where

$$\varphi \equiv (\bigvee_{s \in S_{(q,p)}} x = s) \wedge \bigwedge_{s \notin S_{(q,p)}} x \neq s.$$

- *Case 2.* If $z \in S_{(q,p)}$ for $q, p \in Q$: Let $(q, \varphi, p) \in \delta_p$ where

$$\varphi \equiv \bigwedge_{s \notin S_{(q,p)}} x \neq s.$$

The resulting PA $A_p = (M, Q, q_0, \delta_p, F)$ is equivalent to $\mathcal{A}$. Borrowing notation from PA, the language $L(\mathcal{A})$ corresponds to the union of all $L(\mathcal{A}_\mu)$ where $\mu : \{y_1, \ldots, y_k\} \to D \setminus C$ is an injective map and C denotes the set of constants. The conditions specified in the transition formulae of A_p are implied by the inequality constraints imposed on how variable automata may map their variables to letters, and therefore every word accepted by $\mathcal{A}$ using some μ as defined above can be accepted by A_p using the same μ. This means $L(\mathcal{A}) \subseteq L(A_p)$.

In the other direction, $L(A_p) \subseteq L(\mathcal{A})$ holds if for every word $w \in L(A_p)$, there is an injective parameter assignment $\mu : \{y_1, \ldots, y_k\} \to D \setminus C$ such that $w \in L(\mathcal{A}_\mu)$. Let $w \in L(A_p)$, meaning there is a parameter assignment μ' such that $w \in L((A_p)_{\mu'})$.

If μ' is injective and does not map any variables to C, then the accepting run $(q_0, \varphi_1, q_1), \ldots, (q_{n-1}, \varphi_n, q_n)$ of w in $(A_p)_{\mu'}$ can be mirrored in $\mathcal{A} = \langle D, V \rangle$: In every transition formula φ_i, there is either a clause $x = y_j$ which evaluates to true and corresponds to the transition (q_{i-1}, y_j, q_i) in V or, if no clauses of the form $x = y_j$ exist, the transition (q_{i-1}, z, q_i) in V can be taken. We can thus construct an accepting run in $\mathcal{A}$, assigning the variables according to μ'.

If μ' is not injective or maps variables to C, we need to construct a parameter assignment μ such that $w \in L((A_p)_\mu)$ that is both injective and maps no variables to C, allowing us to fall back to the first scenario.

Assume that there is a variable y_i which is mapped to a constant value $c \in C$. Let $(q_0, \varphi_1, q_1), \ldots, (q_{n-1}, \varphi_n, q_n)$ be an accepting run of w in $(A_p)_{\mu'}$, meaning that all transition formulae φ_j are satisfied. There is therefore no guard φ_j where both the clause $x = y_i$ and the clause $x \neq c$ hold at the same time (or vice versa, the clauses $x = c$ and $x \neq y_i$). Due to the way the guards are constructed, whenever the clause $x = y_i$ is both contained in a guard and evaluates to true, the clause $x = c$ also has to be contained and evaluate to true. We can therefore choose any letter a that is not a part of w or C and modify μ' such that y_i is mapped to a. Using this modified parameter assignment μ'_a, w will still complete an accepting run in $(A_p)_{\mu'_a}$.

The same procedure can be applied if two parameters y_i and y_j are mapped to the same letter. After a finite number of steps, we obtain a parameter assignment μ which is injective and does not map any variables to C. This concludes the proof of $L(A_p) = L(\mathcal{A})$.

- *Register automata*: In example 2.34, VA and RA were shown to be incomparable. Using the same example languages, the statement can be proven for PA and RA.

  The language $L_A = \{w = (w_1, \ldots, w_n) \in D^* \mid w_i \neq w_n \text{ for all } i < n\}$ cannot be identified by an RA, but it can be identified by a VA and therefore, by extension, by a PA.

  The language $L_R = \{w = (w_1, \ldots, w_n) \in D^* \mid w_{2i} = w_{2i-1} \text{ for all } 2i \leq n \text{ and } n \text{ is even}\}$ can be identified by an RA, as seen in figure 2.3. However, L_R cannot be identified by a PA.

  Assume for a proof by contradiction that a PA A identifying L_R exists, having n states and k parameters. Let $w = w_1 w_1 w_2 w_2 \ldots w_j w_j \in L_R$ be a word of length $\geq 3n^2$ such that $w_i \neq w_l$ for all $i \neq l$, i. e., no letter w_i appears in more than one pair. Since $w \in L_R$, $w \in L(A_\mu)$ for some parameter assignment μ. Because A has at most n^2 distinct transitions, w's path through A_μ has to traverse one of these transitions thrice or more, using at least two distinct letters w_i, w_l since no letter occurs more than twice in w. If we replace one occurrence of w_i with w_j, the resulting word is not part of L_R anymore. However, it will still complete the same path in A_μ and terminate in an accepting state. Because of this false-positive result, A cannot identify L_R correctly and a PA identifying L_R cannot exist.

  □

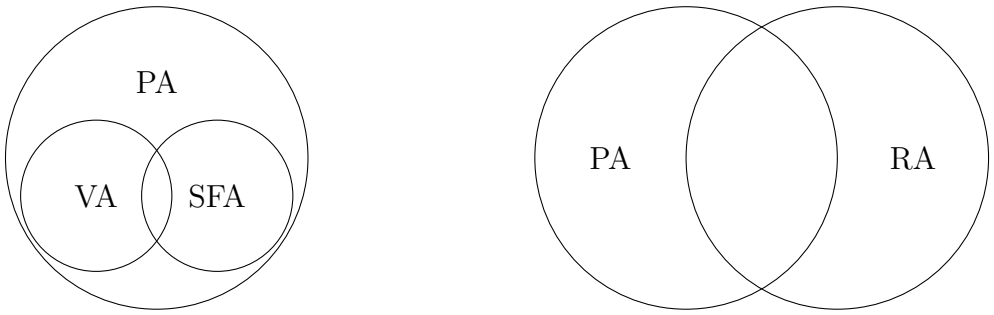


Figure 3.3: Relationships between the sets of languages represented by different automata classes.

3.2 Basic Operations on Parametrized Automata

The definition of parametrized automata does not permit ε-transitions, a concept that is known in both finite-state and symbolic automata. An ε-transition permits a transition between states without consuming a letter and, in the case of SFA, without evaluation of a formula. We will use ε-transitions to construct the union of two parametrized automata, so the concept will be introduced briefly – followed immediately by a proof showing ε-transitions can always be eliminated. This way, we can have our cake and eat it, too, by using ε-transitions whenever they are convenient and omitting them whenever they are clumsy in a proof.

Definition 3.6 (parametrized automata with ε-transitions)**.** A parametrized automaton with ε-transitions (ε-PA) is a tuple $A = (M, Q, q_0, \delta, F)$, where

- D is an infinite alphabet,
- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting states, and
- $\delta \subseteq_{fin} Q \times (\Phi \cup \varepsilon) \times Q$ denotes the transition relation using a set of formulae Φ and the symbol ε, which should not occur in any of the formulae in Φ.

A word $w = (w_1, \ldots, w_k) \in D^*$ is accepted by the automaton A if there exists a sequence of letters $a_1, \ldots, a_n \in D \cup \{\varepsilon\}$, a family of variable assignments $(\mu_i)_{i=1,\ldots,n}$ and a complete run (q_0, φ_1, q_1), (q_1, φ_2, q_2), $\ldots$, $(q_{n-1}, \varphi_n, q_n)$ in A such that

- $a_1 a_2 \ldots a_n = w$,
- for all $i = 1, \ldots, n$, $\varphi_i = \varepsilon \Leftrightarrow a_i = \varepsilon$,
- $\mu_i(y) = \mu_j(y)$ for all $i, j \in \{1, \ldots, n\}$ and $y \in Y$,
- $\mu_i(x) = a_i$ for all $i = 1, \ldots, n$ where $a_i \neq \varepsilon$, i. e., x is assigned the value of the current letter in every step,
- $v_{M,\mu_i}(\varphi_i) = \mathit{true}$ for $i = 1, \ldots, n$, i. e., all involved formulae are true, and
- $q_n \in F$, i. e., the run terminates in an accepting state.

This concept of ε-transitions only permits ε-transitions where no additional guards occur. This means eliminating ε-transitions can be achieved similarly to finite-state automata or symbolic automata, by calculating the ε-closure of states.

Proposition 3.7 (ε elimination)**.** For every PA A with ε-transitions, there is an equivalent PA B (i. e., $L(A) = L(B)$) that does not contain ε-transitions. B can be computed algorithmically.

Proof. Let $A = (M, Q_A, q_0^A, \delta_A, F_A)$ be a PA with ε-transitions. We will construct a version of A without ε-transitions based on [22, section III.1].

For every state $q \in Q$, let $c_\varepsilon(q)$ be the ε-closure of q, i. e., the least subset of Q such that $q \in c_\varepsilon(q)$ and for every $p, r \in Q$ such that $p \in c_\varepsilon(q)$ and $(p, \varepsilon, r) \in \delta$, it follows that $r \in c_\varepsilon(q)$.

Next, for all $q, p \in Q$, let $E(q,p) = \bigvee\{\varphi \mid \exists r : r \in c_\varepsilon(q), (r, \varphi, p) \in \delta, \varphi \neq \varepsilon\} \in \Phi$ be the disjunction of all transition formulas leading from a state in $c_\varepsilon(q)$ to p. Then construct the PA $B' = (M, Q_{B'}, q_0^{B'}, \delta_{B'}, F_{B'})$ with the following properties:

- $Q_{B'} = Q_A$,
- $q_0^{B'} = q_0^A$,
- $\delta_{B'} = \{(q, E(q,p), p) \mid q, p \in Q_{B'}, E(q,p) \in \Phi\}$,
- $F_{B'} = \{q \in Q_{B'} \mid c_\varepsilon(q) \cap F_A \neq \emptyset\}$.

Finally, remove all unsatisfiable transitions and all unreachable states from B' to obtain the automaton B with the desired property. The topic of identification of unreachable states will be treated in more detail in section 3.5. It is plain that B accepts exactly the same words as A does, and that B does not contain any ε-transitions. If A has n states, B will also have n or fewer states. □

There is no clear-cut way to approach determinism in PA. In FSA or SFA, determinism can be defined in two equivalent ways: Each word completes exactly one run if and only if for each reachable state and each letter, there is exactly one viable exiting transition.

This does not hold for PAs. The latter, "local" definition of determinism ensures determinism for a fixed parameter assignment. However, by construction and intention, a word's path through a PA also depends on the chosen parameter assignment. Determinism in PA is a complex subject that can manifest on different levels, which will be explored in more depth in chapter 4. We will introduce a first notion of determinism that is based on the definition of determinism for symbolic automata and allows us to ignore the effect of different parameter assignments.

Definition 3.8 (determinism per assignment)**.** A PA A is deterministic per assignment if for every parameter assignment μ, the SFA A_μ is deterministic.

Determinism per assignment is a useful property. It guarantees a word will always complete a run and its terminating state will be determined by the chosen parameter assignment. We will see later how this is beneficial whenever PA need to be combined. On the flip side, words that are part of the PA's language will frequently terminate in non-accepting states, since words are commonly only accepted by a narrow range of parameters.

Every PA can be transformed into an equivalent PA that is deterministic per assignment.

Proposition 3.9. For every PA A, an equivalent PA B that is deterministic per assignment can be computed algorithmically.

Proof. This proof is heavily based on the algorithms for determinizing SFA [22, section III.2] and FSA [11, section 2.2.1]. An FSA without ε-transitions is determinized by operating on sets of states, instead of the states themselves. A transition (S, a, T) between two sets of states S and T exists if T contains exactly the states that can be reached from states in S via an a-transition. This way, for every letter and every S, there is exactly one exiting transition.

SFA work with logical formulas instead of letters, so the guards of transitions exiting a state-set S need to be constructed in a way to ensure that no two exiting transitions "overlap". Even if two formulas are syntactically distinct, there may be letters such that both formulas evaluate to *true*. For this purpose, sets of transitions are considered: For any set of states S, consider the set of all transitions that exit any state in S. For each letter x, this set of transitions divides cleanly into two subsets, one of which contains all viable transitions while the other contains the remainder, whose guards are not satisfied by x. Let T be the set of states reached by the set of viable transitions, and label a transition between S and T with the conjunction of all formulas satisfied by x, while also adding the conjunction of all *negated* remaining formulas. After proceeding this way

for all subsets of transitions exiting states in S, there will always be exactly one viable transition exiting S for every x.

We formalize this notion and extend it to PA by describing a determinization algorithm.

Let $A = (M, Q_A, q_0^A, \delta_A, F_A)$. For a transition $t = (q, \varphi, p) \in \delta_A$, let $source(t) = q$ be the state it exits, $formula(t) = \varphi$ be its guard and $target(t) = p$ be the state it enters. For a state q, let $L(q) = \{t \in \delta_A \mid source(t) = q\}$ be the set of all transitions exiting q. Similarly, for a set of states S, let $L(S) = \bigcup_{q \in S} L(q)$. Also, extend the definition of $target(t)$ to sets of transitions, such that for a set $T \subseteq \delta_A$, $target(T) = \bigcup_{t \in T} target(t)$.

For the subset construction, let $P(Q_A)$ denote the power set of Q_A, meaning that ultimately $Q_B \subseteq P(Q_A)$. Let G be a stack of elements of $P(Q_A)$ as a frontier (initially, $G = (\{q_0\})$), V a set of elements of $P(Q_A)$ that were already visited (initially, $V = \{\{q_0\}\}$), and T a set of transitions in B, meaning $T \subseteq_{fin} P(Q_A) \times \Phi \times P(Q_A)$ (initially, $T = \emptyset$).

While G is not empty, pop the element S from G. For each subset $L_i \subseteq L(S)$, let $\varphi_{L_i} = \top \wedge (\bigwedge_{t \in L_i} formula(t)) \wedge (\bigwedge_{t \in L(S) \setminus L_i} \neg formula(t))$. If $\varphi(L_i)$ is satisfiable, add $(S, \varphi_{L_i}, target(L_i))$ to T and if $target(L_i)$ has not been visited yet, i. e., $target(L_i) \notin V$, add $target(L_i)$ to both V and G. This first satisfiability check can save runtime, but is not sufficient to ensure that the resulting automaton does not contain unreachable states. Therefore, it can be omitted.

When G is empty, let $B' = (M, V, \{q_0^A\}, T, \{S \in V \mid S \cap F_A \neq \emptyset\})$. Unreachable states and transitions may be removed. Normalize B' by detecting all instances where there are two distinct transitions (S, φ_1, T) and (S, φ_2, T) between two states S and T, and replace those with the single transition $(S, \varphi_1 \vee \varphi_2, T)$.

Since V is a subset of the power set of Q_A, the size of V is bounded by 2^n, where n is the size of Q_A. □

The construction of guards ensures that no two transitions exiting a state can be viable at the same time, no matter the parameter assignment. There is a subtle difference between this algorithm and the one proposed in [22]: The subset L_i may be empty, in which case the transition guard leading to the state $\{\}$ in B is calculated. If $S = \{\}$, then the transition $(\{\}, \top, \{\})$ is calculated. This modification ensures that for every letter and in every state, at least one exiting transition is viable, which was one condition of determinism of symbolic automata.

Notation 3.10. While the described algorithm can lead to a unique B, there may exist several PA that are deterministic per assignment and equivalent to A. Nevertheless, it may be useful to treat this process of determinization like an operation, especially when used in convoluted proofs where it is hard to keep track of the constructed PA. In those situations, we will often denote a PA that is equivalent to A and deterministic per assignment as $\langle A \rangle$.

3.3 Closure Properties

Finite-state automata are closed under union and intersection: Given two arbitrary FSA A and B, we can construct an FSA C such that $L(C) = L(A) \cup L(B)$ and an FSA D such that $L(D) = L(A) \cap L(B)$. If two languages can be represented by FSA, then their union and intersection can also be represented by suitable FSA.

The intersection of two parametrized automata can be constructed using a *product construction*. A product automaton can simulate a word's run through two automata at the same time. A product automaton's function depends on the chosen set of accepting states, therefore we will choose an approach that allows for a separation of the construction of the product from the choice of accepting states. This way, we can make statements about the product of two automata in general without having to restrict ourselves to one specific purpose. We do not speak of "the" direct product, but rather a set of PA that only differ by their accepting states.

Definition 3.11 (direct product construction)**.** Let $A = (M, Q, q_0, \delta_A, F_A)$ and $B = (M, P, p_0, \delta_B, F_B)$ be two PA. Let $Y_A \subset Y$ be the (finite) set of free variables occurring in the guards of δ_A and $Y_B \subset Y$ be the (finite) set of free variables occurring in the guards of δ_B.

In order to allow for both automata to assign their parameters independently, assume without loss of generality that $Y_A \cap Y_B = \varnothing$. Should the free variables (except x) occurring in A and B overlap, the variables of B can always be relabeled via a transformation $f : Y_B \to Y$ such that f is injective and $Y_A \cap f(Y_B) = \varnothing$, $f(Y_B)$ denoting the image of f. Assume that A and B use the same variable x to represent the current letter.

Then we define the direct product $A \times B = \{(M, Q \times P, (q_0, p_0), \delta, F) \mid F \subseteq Q \times P\}$ such that $\delta = \{(q, p), \varphi_1 \wedge \varphi_2, (q', p') \mid (q, \varphi_1, q') \in \delta_A, (p, \varphi_2, p') \in \delta_B\}$. *Any* element of this set can be referred to as *a* direct product of A and B.

A weakness of this approach to product automata is that the set $A \times B$ does not retain any information on the accepting states of A and B. This information has to be transferred and converted manually, depending on our goals. For a PA $C \in A \times B$, let F_C be its set of accepting states. Since F_C is the only trait setting C apart from other PA in $A \times B$, we introduce the shorthand notation $C = (A \times B, F_C)$.

A run of a word in a direct product automaton is fully determined by the run of the word in the original automata, and vice versa.

Theorem 3.12. Let $A = (M, Q, q_0, \delta_A, F_A)$ be a PA using the finite set of parameters Y_A, and $B = (M, P, p_0, \delta_B, F_B)$ be a PA using the finite set of parameters Y_B where $Y_A \cap Y_B = \varnothing$. Let $C = (A \times B, F_C)$ be a direct product automaton. Then a run of a word w in C_μ (where $\mu : Y_A \cup Y_B \to D$ is a parameter assignment) terminates in the state (q, p) if and only if a run of w in $A_{\mu|_{Y_A}}$ terminates in the state q and a run of w in $B_{\mu|_{Y_B}}$ terminates in state p. Here, $\mu|_{Y_A}$ and $\mu|_{Y_B}$ denote the restriction of μ to the sets Y_A and Y_B, respectively.

Proof. Assume that w terminates in the state $(q, p) \in Q \times P$ in C_μ. Then there is a sequence of states and transitions $((q_0, p_0), \varphi_0, (q_1, p_1)), \ldots, ((q_n, p_n), \varphi_n, (q, p))$ such that $v_{\mu,w_i}(\varphi_i) = \mathit{true}$ for $i = 0, \ldots, n$, where v_{μ,w_i} denotes the evaluation function when plugging in the current letter w_i and the parameter values specified by μ.

For each transition $((q_i, p_i), \varphi_i, (q_{i+1}, p_{i+1}))$, the formula φ_i consists, by definition, of two subformulae $\psi_i^1 \wedge \psi_i^2 \equiv \varphi_i$ such that $(q_i, \psi_i^1, q_{i+1}) \in \delta_A$ and $(p_i, \psi_i^2, p_{i+1}) \in \delta_B$. Since $v_{\mu,w_i}(\varphi_i) = \mathit{true}$ if and only if $v_{\mu,w_i}(\psi_i^1) = \mathit{true}$ and $v_{\mu,w_i}(\psi_i^2) = \mathit{true}$, we can conclude that w completes the run $(q_0, \psi_0^1, q_1), \ldots, (q_n, \psi_n^1, q)$ in A_μ and the run $(p_0, \psi_0^2, p_1), \ldots, (p_n, \psi_n^2, p)$ in B_μ. The parameters Y_B do not occur in guards of A and can therefore be omitted, and vice versa.

Now assume a word w completes a run in A_{μ_A} in a state q and a run in B_{μ_B} in a state p (where $\mu_A : Y_A \to D$ and $\mu_B : Y_B \to D$ are some parameter assignments). Since Y_A and Y_B are disjoint sets, there is a well-defined parameter assignment $\mu : Y_A \cup Y_B \to D$, where

$$\mu(y) = \begin{cases} \mu_A(y) & \text{if } y \in Y_A \\ \mu_B(y) & \text{if } y \in Y_B. \end{cases}$$

Obviously, w completes a run in C_μ in state (q, p). □

Direct product automata are capable of simulating the behaviour of two automata independently because of the demand that the sets of parameters do not intersect.

Later, we will need a type of product construction where the parameters are, in fact, dependent. We will call this construction the *synchronized product*. For now, we will only need to work with the direct product.

Corollary 3.13. The direct product of two PA that are deterministic per assignment is always deterministic per assignment.

Conveniently, the intersection of two PA can be computed using a direct product automaton that accepts a word if and only if it is accepted by both parent automata.

Corollary 3.14 (intersection of PA)**.** For two PA A and B, a PA C such that $L(C) = L(A) \cap L(B)$ can always be computed. If A has n states and i parameters and B has m states and j parameters, the resulting PA will have $\mathcal{O}(nm)$ states and $i + j$ parameters.

Proof. Let $C = (A \times B, F_\cap)$ be a direct product automaton of A and B such that $F_\cap = F_A \times F_B$. A word is accepted by C if and only if it is accepted by both A and B. □

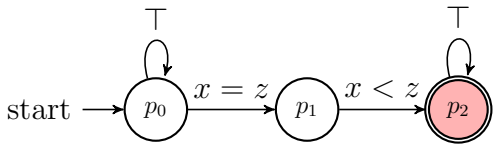
An example of the union and intersection of PA can be found in figure 3.4.

A direct product construction is not useful for constructing the union of two PA: If a word is accepted by an automaton A but never completes a run in an automaton B, it will never complete a run in any direct product automaton of $A \times B$ despite being part of the union $L(A) \cup L(B)$. We could transform A and B into equivalent PA that are deterministic per assignment first, however this may lead to an exponential blowup of the number of states. A more efficient alternative is to connect the two automata using an ε-transition:

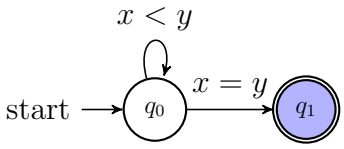
Proposition 3.15 (union of PA)**.** For any two PA A and B, there is an ε-PA C such that $L(C) = L(A) \cup L(B)$. If A has n states and B has m states, then C will have $\mathcal{O}(n + m)$ states.

Proof. Let $A = (M, Q, q_0, \delta_A, F_A)$ and $B = (M, P, p_0, \delta_B, F_B)$. Assume without loss of generality that Q and P, as well as δ_A and δ_B, are disjoint. Let r be a state not in Q or P. Then the automaton $C = (M, Q \cup P \cup \{r\}, q, \delta_A \cup \delta_B \cup \{(r, \varepsilon, q_0), (r, \varepsilon, p_0)\}, F_A \cup F_B)$ satisfies $L(C) = L(A) \cup L(B)$.

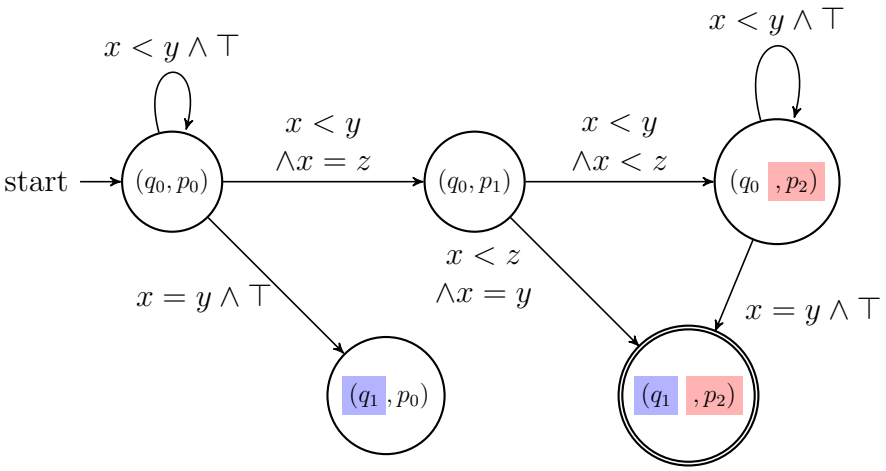
To prove that $L(C) \subseteq L(A) \cup L(B)$, let w terminate a run in an accepting state of C. Assume without loss of generality that w terminates in a state in F_A. Then w also terminates a run in A in F_A, since the initial ε-transition does not consume a letter of the word. Since the same goes for runs terminating in F_B, we can conclude $L(C) \subseteq L(A) \cup L(B)$.



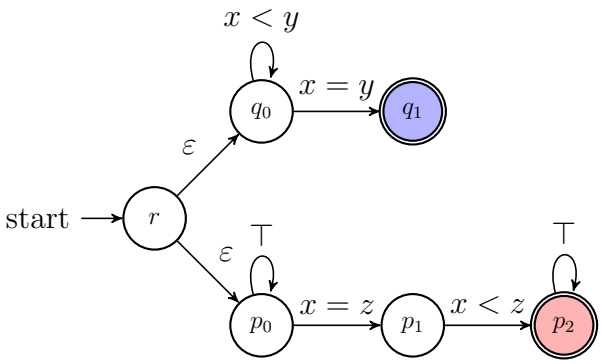
(a) A_1, with states and parameters relabeled to avoid ambiguity.



(b) The PA A_2.



(c) The direct product automaton of A_1 and A_2 identifying the intersection.



(d) Union of A_1 and A_2.

Figure 3.4: Union and intersection of PA.

To prove that $L(A) \cup L(B) \subseteq L(C)$, assume without loss of generality that $w \in L(A)$. That means w terminates a run $(q_0, \varphi_0, q_1), \ldots, (q_{k-1}, \varphi_{k-1}, q_n)$ in an accepting state $q_n \in F_A$. Therefore, w also terminates the run $(r, \varepsilon, q_0), (q_0, \varphi_0, q_1), \ldots, (q_{k-1}, \varphi_{k-1}, q_n)$ in C in an accepting state. In conclusion, $L(A) \subseteq L(C)$. In the same manner, $L(B) \subseteq L(C)$ can be proven.

Therefore, $L(C) = L(A) \cup L(B)$. □

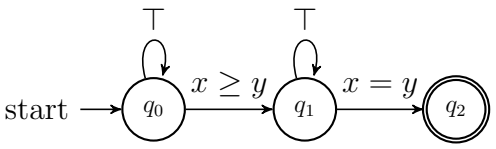
This method of constructing the union does not preserve determinism per assignment and introduces ε-transitions, which we may not desire. Therefore, a direct product construction can still be useful for computing the union in some cases.

Corollary 3.16. When two PA $A = (M, Q, q_0, \delta_A, F_A)$ and $B = (M, P, p_0, \delta_B, F_B)$ are deterministic per assignment, the direct product $(A \times B, F_A \times F_B)$ expresses their union.

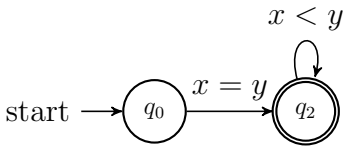
Remark 3.17. PA are closed under reversal and concatenation.

Proof. We "borrow" this observation from the subsequent section on complementation of PA: It is indirectly proven by proposition 3.21. □

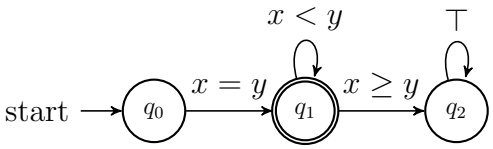
3.4 Complementable Parametrized Automata



(a) A_2^c, which identifies the complement of $L(A_2)$.



(b) A_2^R, which identifies the reverse of $L(A_2)$.



(c) D_2^R, a strongly deterministic variant of A_2^R.

Figure 3.5: Different automata for the complement and reversal of $L(A_2)$.

Not all PA can be complemented. This section is therefore dedicated to the subset of PA that can be complemented, and will contain an investigation of their closure properties.

Definition 3.18 (complementable PA)**.** A PA A is complementable if there exists another PA A^c such that $w \in L(A) \Leftrightarrow w \notin L(A^c)$.

Example 3.19. The PA A_2 and A_3 from figure 3.1 are complementable. Examples of their complement automata can be seen in figure 3.5 and figure 4.5, respectively.

A_1 is not complementable, as seen in theorem 3.4.

The complement of a complementable PA is obviously also complementable. The same holds for the union and intersection, which is only slightly less trivial:

Proposition 3.20 (closure of complementable PA under Boolean operations)**.** Let A and B be two complementable PA with complements A^c and B^c, respectively. Then the PA representing their union and intersection, here denoted $A \cup B$ and $A \cap B$, are also complementable.

Proof. Using the known complement automata, De Morgan's laws ([1, section 1.6]) can be extended to PA:

$$\begin{aligned} w \in L(A^c \cap B^c) &\Leftrightarrow w \in L(A)^c \cap L(B)^c \\ &\Leftrightarrow w \in (L(A) \cup L(B))^c \\ &\Leftrightarrow w \in L(A \cup B)^c, \end{aligned}$$

where $A \cup B$ denotes the union automaton of A and B. Define $(A \cup B)^c = A^c \cap B^c$.

Analogously,

$$\begin{aligned} w \in L(A^c \cup B^c) &\Leftrightarrow w \in L(A)^c \cup L(B)^c \\ &\Leftrightarrow w \in (L(A) \cap L(B))^c \\ &\Leftrightarrow w \in L(A \cap B)^c, \end{aligned}$$

where $A \cap B$ denotes the intersection automaton of A and B. Define $(A \cap B)^c = A^c \cup B^c$. □

Proposition 3.21 (reversal of complementable PA)**.** Complementable PA are also closed under reversal.

A language's reversal consists of all words of the language "read backwards", and can be defined inductively for words as $a^R = a$ for letters a and $(aw)^R = w^R a$ if the suffix w is a word. An automaton representing a language can be reversed by reversing all arrows, marking the former initial state as accepting and turning all former accepting states into initial states (if the original automaton has more than one accepting state, we can instead introduce a new initial state and connect it to all former accepting states using ε-transitions).

Proof. The reversal of a complement of a language equals the complement of the reversal, i. e., a complementable PA's reversal can be complemented by reversing the complement. Let A be a PA with complement automaton A^c.

$$\begin{aligned} w \in (L(A)^C)^R &\Leftrightarrow \exists v \in L(A)^C : v = w^R \\ &\Leftrightarrow w^R \notin L(A) \\ &\Leftrightarrow w \notin L(A)^R \\ &\Leftrightarrow w \in (L(A)^R)^C. \end{aligned}$$

□

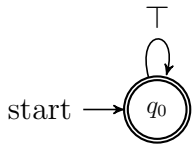
In summary, complementable PA are closed under all Boolean operations and reversal. The nice closure properties start to break down, however, when investigating concatenation.

Concatenation of two languages can be performed directly on the PA representing the languages. Let $A = (M, Q, q_0, \delta_A, F_A)$ and $B = (M, P, p_0, \delta_B, F_B)$ be two PA, where the sets Q and P are disjoint. Let $\delta^* = \{(q, \varepsilon, p_0) \mid q \in F\}$ be a set of ε-transitions connecting all accepting states of A to the initial state of P. Then the automaton $AB = (M, Q \cup P, q_0, \delta \cup \delta^*, F_B)$ identifies the concatenation of the languages of A and B, $L(AB) = L(A)L(B)$.

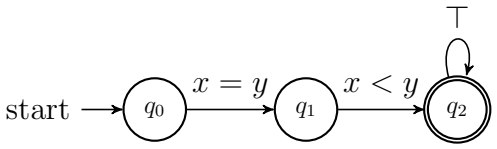
Proposition 3.22. Complementable PA are not closed under concatenation.

Proof. As confirmed by theorem 3.4, the complement of the language L_1 of all unsorted words cannot be recognized by a PA. However, L_1 can be represented by the concatenation of two complementable PA.

Each unsorted word has a suffix w whose first letter is larger than the second. The language in which the first letter is larger than the second letter can be identified by a PA A that is complementable. In order to represent all words from L_1, the occurrence of this instance of disorder can be offset by allowing arbitrary words (stemming from the universal language, represented by the PA U) to be attached in front. The concatenation of U and A therefore identifies L_1. □



(a) A PA U for the universal language, which can be complemented by marking state q_0 as non-accepting.



(b) A PA A accepting all words in which the first letter is larger than the second letter.

Figure 3.6: Two example PA that, when concatenated, are equivalent to A_1.

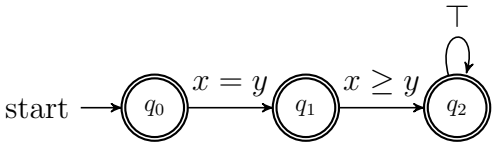


Figure 3.7: A complement automaton for A, proving A can be complemented.

3.5 Decision Problems

The following section is dedicated to the most common decision problems regarding parametrized automata. Similar to our cursory investigation of FSA, we will look at the universality, equivalence, containment, non-emptiness and reachability problems. The universality problem will quickly turn out to be undecidable for PA, which has dire consequences for problems relating to the complementation of PA.

Theorem 3.23 (universality)**.** Universality is undecidable for PA.

Proof. Universality is undecidable for variable automata (see [10, chapter 4]), as can be shown by a reduction from Post's Correspondence Problem. For PA in general, we can either do a similar reduction or argue, since PA subsume VA, that undecidability follows directly. □

Corollary 3.24 (equivalence)**.** The question whether two PA A and B are equivalent, i. e., $L(A) = L(B)$, is undecidable.

Proof. Assume that there is a Turing machine M determining whether two input PA A and B are equivalent. We can use M to find out whether an arbitrary PA A is universal, by setting B to a PA representing the universal language. This contradicts the undecidability of universality. □

Corollary 3.25 (containment)**.** The containment problem for PA (i. e., for two PA A and B, is $L(A) \subseteq L(B)$?) is undecidable.

Proof. Assume that there is a Turing machine M determining whether $L(A) \subseteq L(B)$ for two input PAs A and B. We can use M to find out whether an arbitrary PA B is universal, by setting A to a PA representing the universal language. This contradicts the undecidability of universality. □

The non-emptiness problem and the reachability problem are decidable (see also [13, section 3, proposition 1]).

Theorem 3.26 (non-emptiness, reachability)**.** Let T be a theory. Assume that satisfiability of T-formulas is decidable in NP [PSPACE]. Then the non-emptiness problem and the problem of deciding whether a certain state of an automaton is reachable are also decidable for all PA using T in NP [PSPACE].

Proof. Starting with the reachability problem: To check whether a single state q is reachable, it is sufficient to list all non-cyclic paths from the initial state to q and check if there exists a word w and a parameter assignment μ such that w can traverse one of these paths. For a path $(q_0, \varphi_0, q_1), (q_1, \varphi_1, q_2), \ldots, (q_m, \varphi_m, q)$, the variable x occurring in each φ_i needs to be relabeled to a distinct x_i (where x_i is no parameter and not equal to any other x_j), since the letters of the word can and should be chosen freely. The parameters $\{y_1, \ldots, y_k\}$ need not be relabeled.

Thus, the conjunction $\bigwedge_{i=0}^{m} \varphi_i$ contains the free variables $x_0, \ldots, x_m, y_1, \ldots, y_k$. The conjunction is satisfiable if and only if there exists a word $w = x_1 x_2 \ldots x_m \in D^*$ and a parameter assignment $\mu : Y \to D$ such that w traverses the path $(q_0, \varphi_0, q_1), (q_1, \varphi_1, q_2), \ldots, (q_m, \varphi_m, q)$, terminating in q. Since the satisfiability check is decidable and there is a finite number of non-cyclic paths from the initial state to q, reachability is decidable.

A PA is empty if and only if none of its accepting states are reachable. Therefore, we simply need to perform a reachability check on all accepting states. □

At this point, we can briefly describe the removal of unnecessary transitions. The identification of unreachable states has already been discussed, so those can be removed (including all transitions leading to said states) without altering the functionality of the PA itself.

The identification of unsatisfiable transitions in symbolic automata is fairly simple, as it is sufficient to consider a guard in isolation and determine its satisfiability without further context. In parametrized automata, a local examination of guards may not be sufficient because prior choices of parameters can influence the satisfiability of guards. While a formula in isolation may be satisfiable, it may be unsatisfiable when a number of guards on a path leading to said formula have to hold at the same time.

Let (q, φ, p) be a transition, meaning that φ is the guard we want to check for satisfiability. Similar to the reachability check, all non-cyclic paths leading from the initial state q_0 to q can be listed. After relabeling the x-variables in the guards $\varphi_1, \varphi_2, \ldots, \varphi_m$ along a path, we can construct a similar conjunction $\bigwedge_{i=0}^{m} \varphi_i$. The transition φ is globally satisfiable only if there exists a path such that $\bigwedge_{i=0}^{m} \varphi_i \wedge \varphi$ is satisfiable. If no such path exists, the transition is never viable and can be removed without consequences.

Figure 3.8: The universal language and its complement, expressed by PA.

Complementation can be used as a tool in deciding universality of PA: If a PA cannot be complemented, it cannot be universal as the complement of the universal language can be represented by a PA (see figure 3.8). If a PA can be complemented and its complement is empty, as a consequence the original PA has to be universal.

However, the universality problem is undecidable for PA, and therefore, the complementation problem cannot be "easy".

Corollary 3.27 (undecidability of complementation)**.** Under the condition of non-emptiness being decidable, there is no Turing machine which can take an arbitrary PA as input and then either returns its complement, or returns that a complement does not exist.

Proof. Assume that there is a Turing machine M which takes a PA as input and returns either a PA describing the complement, or returns that such a PA does not exist. If a PA A cannot be complemented, it cannot be universal. If it is complementable, the complement can be checked for non-emptiness. If the complement is empty, the language is universal. This contradicts undecidability of universality of PA. □

There is no hope of finding a practical "two-in-one"-algorithm that both identifies complementable PA, and then computes the complement. It remains an open question whether both steps are undecidable, or if maybe one part of the process is decidable.

The following chapters will therefore be dedicated to partial solutions and approximations, identifying subclasses of PA that are easier to complement. The first will be strongly deterministic parametrized automata.

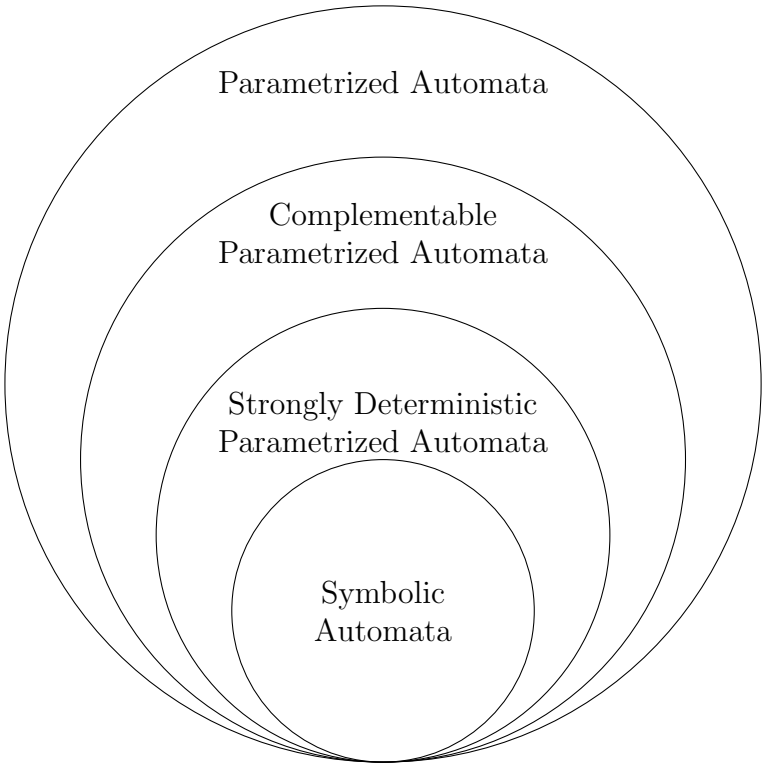
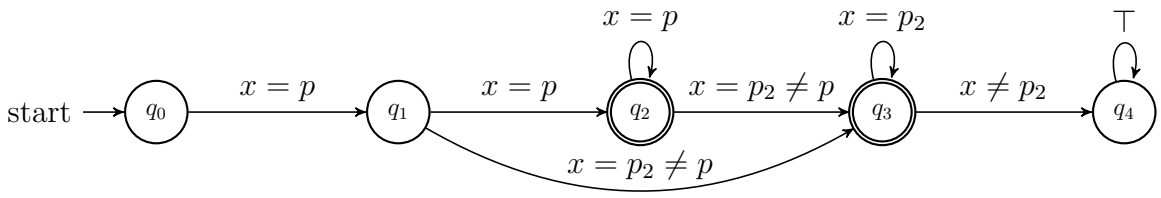


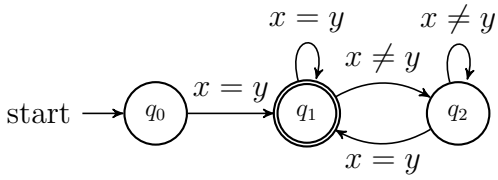
Figure 3.9: Relations between the different types of parametrized automata.

Chapter 4

Strongly Deterministic Parametrized Automata: A Complementable Fragment



(a) An SDPA equivalent to P from section 1.1.



(b) D_4, identifying all words whose first and last letter coincide.

Figure 4.1: Two examples of strongly deterministic parametrized automata.

4.1 Definition and General Properties

Finite-state automata and symbolic automata use a notion of determinism that allows for easy complementation: In a deterministic FSA or SFA, every word has exactly one complete run, so a word is part of the complement language if and only if its run terminates in a non-accepting state.

Determinism per assignment has been introduced in definition 3.8. In a PA that is deterministic per assignment, there is exactly one complete run for each word – assuming the automaton is restricted to one fixed parameter assignment. A word's run may still terminate in different states for different parameter assignments. Therefore, determinism

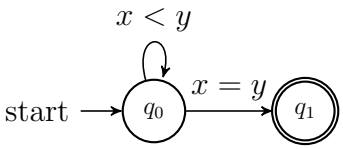


Figure 4.2: A_2, as seen in figure 3.1.

per assignment is not a suitable foundation for complementation algorithms as seen in FSA or SFA.

In this regard, PA share similarities with variable automata ([10]). For VA, a deterministic pattern automaton does not guarantee each word to complete exactly one run in the corresponding VA. In search for a more meaningful notion of determinism that ideally paves the way for complementation, the authors of [10] have therefore proposed a definition of determinism that demands each word to have exactly one run considering all possible variable assignments. This way, since no word that is part of the language may ever terminate in a non-accepting state, deterministic variable automata can be complemented as easily as deterministic finite-state or symbolic automata.

In this chapter, we will investigate whether this approach is suitable for parametrized automata.

Definition and Notation 4.1 (sets of runs)**.** For a PA $A = (M, Q, q_0, \delta, F)$, let $\Gamma(A)$ denote the set of all finite sequences $s = (q_0, \varphi_0, q_1), (q_1, \varphi_1, q_2), \ldots, (q_n, \varphi_n, q_{n+1})$ starting in the initial state q_0 and terminating in some arbitrary state $q_{n+1} \in Q$ where each $(q_i, \varphi_i, q_{i+1}) \in \delta$ is a transition of A. Note that runs in $\Gamma(A)$ may contain transition formulae that are, in conjunction, unsatisfiable; we only demand each transition to begin in the state entered by the previous transition of the sequence. Intuitively, $\Gamma(A)$ contains all candidates for runs in A.

Let $w \in D^*$ be a word and $\mu \in \Theta$ be a parameter assignment. Each run of w in A_μ corresponds to a unique sequence of transitions in $\Gamma(A)$.

Then we define $p_\mu(w) \subseteq \Gamma(A)$ to be the set of all sequences of transitions representing complete runs of w in A_μ, and $p^a_\mu(w) \subseteq p_\mu(w) \subseteq \Gamma(A)$ to be the set of all sequences of transitions representing *accepting* runs of w in A_μ.

For a set S, let $|S|$ be the number of elements of S. If S is infinite, define $|S| = \infty$. Else we denote $n = |S| < \infty$.

Example 4.2. Consider A_2 from figure 4.2. There are two transitions in A_2, $t_1 = (q_0, x < y, q_0)$ and $t_2 = (q_0, x = y, q_1)$. Using regular expression notation as introduced in definition 2.7, $\Gamma(A_2)$ can be described by the expression $t_1^*(t_2 + \varepsilon)$, i. e., $\Gamma(A_2)$ contains all sequences of transitions where t_1 repeats an arbitrary number of times, followed by one or zero occurrences of t_2.

Let $w = (1, 2, 3, 4) \in L(A)$, and $\mu : \{y\} \to D^*$ be a parameter assignment. Then

$$p_\mu(w) = \begin{cases} \emptyset & \text{if } \mu(y) < 4 \\ \{t_1 t_1 t_1 t_2\} & \text{if } \mu(y) = 4 \\ \{t_1 t_1 t_1 t_1\} & \text{if } \mu(y) > 4 \end{cases}$$

and $|p^a_\mu(w)| = 1 \Leftrightarrow \mu(y) = 4$.

Proposition 4.3. For every PA $A = (M, Q, q_0, \delta, F)$, the set $\Gamma(A)$ is countable. If A does not contain ε-transitions, $|p_\mu(w)| < \infty$ for every word $w \in D^*$ and every parameter assignment μ.

Proof. We will prove the first statement by finding an injective map $m : \Gamma(A) \to \mathbb{N}$. Let $n = |Q|$ be the number of states of A. Since A is assumed to be normalized, there are no more than n runs in $\Gamma(A)$ of length 1, no more than n^2 runs of length 2, and no more than n^k runs of length $k \in \mathbb{N}$. Enumerate the states $Q = \{q_1, q_2, \ldots, q_n\}$. A run $((q_0, \varphi_{i_1}, q_{i_1}), \ldots, q_{i_{k-1}}, \varphi_{i_k}, q_{i_k})$ of length k, where $i_1, \ldots, i_k \in \{1, \ldots, n\}$, can be characterized solely by the sequence of states $(q_{i_1}, q_{i_2}, \ldots, q_{i_k})$ entered in each transition.

The run of length 0 is mapped to 0. The remaining elements of $\Gamma(A)$ can be enumerated by mapping each sequence $(q_{i_1}, q_{i_2}, \ldots, q_{i_k})$ according to the formula

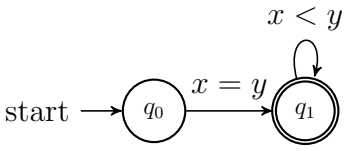
$$\sum_{j=1}^{k} i_j \cdot n^{j-1}.$$

The procedure is visualized in the following table:

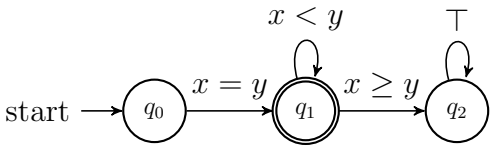
sequence		(q_1)	(q_2)	...	(q_n)	(q_1, q_1)	...	(q_n, q_n)	(q_1, q_1, q_1)	...
$\mathbb{N}$	0	1	2	...	n	$1 + n$	...	$n + n \cdot n$	$1 + n + n \cdot n$	...

We have over-approximated the target function m, since our instructions provide a bijection between $\mathbb{N}$ and the set of all finite sequences over $\{q_1, q_2, \ldots, q_n\}$. Be removing all sequences that are not part of $\Gamma(A)$, we remain with an injective map $m : \Gamma(A) \to \mathbb{N}$.

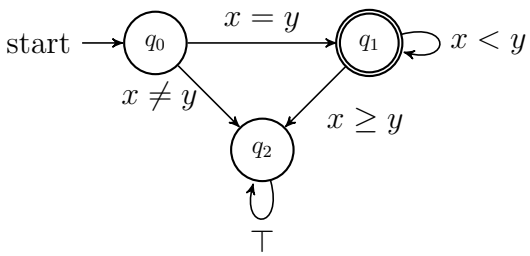
The second statement is a consequence of previous observations: A complete run of a word of length k will traverse exactly k transitions, putting an upper limit of n^k to the number of possible complete runs. □



(a) A_2^R, the reversal of A_2.



(b) An SDPA equivalent to A_2^R.



(c) A PA equivalent to A_2^R that is deterministic per assignment.

Figure 4.3: Different representations for L_2^R, the language of words in which the first letter is strictly largest.

Definition 4.4 (completeness, conclusiveness, strong determinism)**.** Consider the PA $A = (M, Q, q_0, \delta, F)$.

- A is called complete if for every word $w \in D^*$, $|\bigcup_{\mu \in \Theta} p_\mu(w)| \geq 1$, i. e., there is a parameter assignment μ such that the word completes a run.
- A is called conclusive if for every word $w \in D^*$, $|\bigcup_{\mu \in \Theta} p_\mu(w)| \leq 1$, i. e., there is at most one run for every word.
- A is strongly deterministic (an SDPA) if for every word $w \in D^*$, $|\bigcup_{\mu \in \Theta} p_\mu(w)| = 1$, i. e., there is exactly one run for each word.

An SDPA is therefore both complete and conclusive. Note that conclusiveness does not that imply there is at most one parameter assignment for each word to complete a run. It only means that if there are two parameter assignments μ_1 and μ_2 such that a word w completes a run in A_{μ_1} and A_{μ_2}, the runs have to be identical. An example is A_3 as seen in figure 4.3.

We say that a PA A is determinizable if there exists an SDPA A' such that $L(A) = L(A')$.

Example 4.5. Consider the PA depicted in figure 4.3. The first, A_2^R, is conclusive because each state has exactly one exiting transition, forcing each run to either take that transition or die. However, A_2^R is not complete because words that are not part of the language will never complete a run.

When adding a sink state in order to "catch" dying runs, the resulting automaton seen in figure 4.3b is both conclusive and complete and therefore an SDPA.

PA that are deterministic per assignment, as the one in figure 4.3c, are always complete. However, they are usually not conclusive, as different parameter assignments lead to different runs. This phenomenon will be explored in more depth later.

Remark 4.6. Determinism per assignment can be defined retroactively in terms of the notation introduced for SDPA. A PA is deterministic per assignment if for every word $w \in D^*$ and every parameter assignment μ, $|p_\mu(w)| = 1$.

Likewise, completeness and conclusiveness "per assignment" can be defined: A PA A is complete per assignment if for every word w and every parameter assignment μ, $|p_\mu(w)| \geq 1$. A PA A is conclusive per assignment if for every word w and every parameter assignment μ, $|p_\mu(w)| \leq 1$.

Next, the properties of SDPA should be investigated. Similar to the course of action regarding other classes of automata previously covered in this thesis, we will have a look at the closure properties under Boolean operations first. Fortunately, we do not need any more intermediate steps or more refined tools to obtain the desired results: The direct product construction that has been introduced in section 3.3 is sufficient to prove that SDPA are closed under union and intersection operations. Since SDPA have been introduced with complementation in mind, it is trivial to prove that SDPA are also closed under complementation.

Corollary 4.7. The direct product of two conclusive PA is also conclusive. The direct product of two complete PA is also complete.

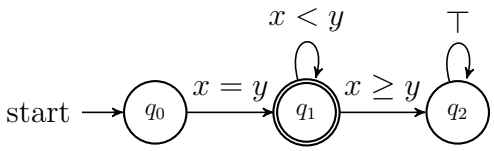


Figure 4.4: D_2, identifying all words in which the first letter is largest, as seen in figure 4.3b.

Proof. This is a direct consequence of theorem 3.12. □

Theorem 4.8 (closure of SDPA under Boolean operations)**.** The set of languages represented by SDPA is closed under union, intersection and complementation.

Proof. The direct product set $A \times B$ of two SDPA $A = (M, Q, q_0, \delta_1, F_1)$ and $B = (M, P, p_0, \delta_2, F_2)$ can be constructed as described in definition 3.11. To make the direct product identify the intersection of $L(A)$ and $L(B)$, set the set of accepting states to $F_\cup = F_1 \times F_2$. To make the direct product identify their union instead, choose $F_\cap = (Q \times F_2) \cup (F_1 \times P)$. Because each word has exactly one complete run in A resp. B, it will have one complete run in $(A \times B, F_\cup)$ or $(A \times B, F_\cap)$ with the desired outcome.

To construct the complement of a SDPA A, simply turn all accepting states into non-accepting states and all non-accepting states into accepting states. The resulting automaton A^c identifies $L(A)^c$, because

$$\begin{aligned} w \in L(A) &\Leftrightarrow w \text{ terminates in an accepting state of } A \\ &\Leftrightarrow w \text{ terminates in a non-accepting state of } A^c \\ &\Leftrightarrow w \notin L(A)^c. \end{aligned}$$

□

A direct product of two PA is constructed in quadratic time and, in the worst case, causes a quadratic increase in the number of states. The complement can be constructed in linear time.

Another minor result concerning the closure properties of SDPA is the following:

Proposition 4.9. Strongly deterministic PA are not closed under reversal.

Proof. Consider A_2, which identifies all words whose last letter is largest. In figure 4.4, the reversal of $L(A_2)$ is expressed using an SDPA. $L(A_2)$ itself, however, cannot be expressed using an SDPA.

Assume for contradiction that such an SDPA A does exist and has n states. This prove will make use of the observation that in every SDPA, whenever two words share a prefix, the runs of the prefix have to be identical even if the words complete their respective runs using different parameter assignments. Consider the sequence of words $((1, \ldots, k))_{k \in \mathbb{N}} = (1), (1, 2), (1, 2, 3), \ldots$. For every $i = 1, 2, \ldots$, let μ_i be a parameter assignment such that A_{μ_i} accepts the word $(1, \ldots, i)$.

Then all prefixes of a word $(1, \ldots, i)$ will also complete runs in A_{μ_i}, and the runs need to terminate in accepting states because A is assumed to be strongly deterministic. Obviously, the terminating state q of the run of $(1, \ldots, i)$ has to be distinct from *all* states

traversed by the prefix $(1, \ldots, i-2)$: Otherwise, there would be a transition (q, φ, p) exiting q and entering an accepting state p that is viable for some $j < i$, implying the word $(1, \ldots, i, j)$ to complete an accepting run in A despite not being a part of $L(A_2)$.

Applying this observation to all prefixes of $(1, \ldots, i)$, A_{μ_i} must have at least $\frac{i}{2}$ states. This leads to a contradiction as soon as $i > 2n$. □

Remark 4.10. As a side note, the reversal result also holds for deterministic variable automata (see the language of all words whose first letter appears more than once). Since variable automata are not the focus of this work, the full proof will be omitted.

Aside from closure properties, we are interested in how SDPA hold up when measured against the usual, important decision problems. The universality, containment and equivalence problems are undecidable for parametrized automata, so we hope for an improvement when restricted to SDPA. Indeed, all of these problems are decidable for SDPA and the corresponding proofs are concise.

Theorem 4.11 (universality)**.** Universality is decidable for SDPA.

Proof. An SDPA is universal if and only if all reachable states are accepting. An algorithm for identifying reachable states is described in theorem 3.26, and the identified states can easily be checked for non-acceptance. □

Theorem 4.12. For two PA A and B, the containment problem $L(A) \subseteq L(B)$ is decidable if B is strongly deterministic.

Proof. Since B is strongly deterministic, we can construct the complement automaton B^c. $L(A) \subseteq L(B) \Leftrightarrow L(A) \cap L(B^c) = \emptyset$. □

Corollary 4.13. For two SDPA A and B, the equivalence problem $L(A) = L(B)$ is decidable.

Proof. The containment problem is decidable, and $L(A) = L(B) \Leftrightarrow L(A) \subseteq L(B) \wedge L(B) \subseteq L(A)$. □

4.2 Applicability

The first section of this chapter suggests SDPA may be an attractive fragment of PA not just with respect to complementation, but harder decision problems as well. However, SDPA have been introduced in a very descriptive (rather than constructive) manner and most proofs have been straightforward and effortless, therefore robbing us of the opportunity to learn more about the limitations of SDPA. We will only learn more about these limitations by conducting a manual search.

In order to find out to what degree SDPA are actually applicable to the complementation problem, we need to take a closer look at some of their more specific properties. This is in stark contrast to the more general results of the first section.

This section will start by answering questions related to the *prevalence* of SDPA, and how they fit in the greater context of classes of PA encountered so far. Then, we will investigate to which extent the approach of

1. determinizing a given PA,

2. then complementing the resulting SDPA

makes for a viable strategy to complement arbitrary PA. For this purpose, we need to solve two computational problems: The determinization process itself, and a method that confirms whether a given PA is strongly deterministic.

Parametrized automata combine the properties of both symbolic automata and variable automata, two very different approaches to languages over infinite alphabets and therefore widely incomparable. For this reason, we need to pay greater attention than usual to different notions of determinism, and use this opportunity for a brief review.

To elaborate, consider finite-state automata. An FSA is deterministic if every word completes exactly one run (see definition 2.22). This definition is high-level and descriptive, or semantic, meaning that it merely posits *what* property a deterministic FSA should satisfy, instead of *how* a deterministic FSA should work. It describes the end goal without explaining how to achieve said goal. A definition answering the *"how"* instead of only the *"what"* is more valuable from a computational perspective, because it paves the way for both understanding determinism and utilizing the concept in proofs to our advantage.

In the case of FSA, there is also a local, constructive, syntactic definition of determinism stating an FSA is deterministic if and only if for every state and every letter, there is exactly one exiting transition. This definition is local in the sense that each state can be checked for determinism independently to obtain the greater picture; determinism in FSA is *not* more than the sum of its parts. Both definitions of determinism are equivalent in FSA, speaking for determinism in FSA being straightforward and intuitive. There is no alternative definition for determinism in FSA that differs from our definition in a meaningful way (barring the question whether deterministic FSA should be complete, which is up to convention), and concepts such as ambiguity (only every accepted word needs to complete exactly one run, see [12]) defer to determinism rather than compete with it.

Symbolic automata are comparable: The syntactic definition of determinism (for each combination of a state and a letter, exactly one exiting transition has to be viable) and the semantic definition (each word has to complete exactly one run) describe the same property from different perspectives.

In variable automata, however, this approach starts to break down. The authors of [10], who propose a semantic definition of determinism where each word completes exactly one run[1], quickly observe a VA can be non-deterministic even if its pattern automaton (an FSA) is deterministic. The added layer of different variable assignments permits a word to have different runs regardless. Determinism of the VA is obtained by "culling" the VA, since a word that is accepted cannot be allowed to ever complete a run in a non-accepting state. If the chosen variables do not lead the run of a word to an accepting state, the run has to die.

Therefore, determinism of a VA and determinism of the pattern automaton rarely coincide. Even if they coincide, this implies the variable assignment does not meaningfully influence the VA's operations and the entire automaton can be replaced by some FSA without the hassle of variables in the first place.

[1]The authors later introduce a syntactic definition, which relies on the tight parameter control provided by VA and is therefore not applicable to PA.

Regarding PA, we have introduced two different notions of determinism. Determinism per assignment is loosely based on determinism in SFA while strong determinism is based on determinism in VA. We can and should examine how both concepts relate to one another, and will observe similarities to the traits of determinism in VA.

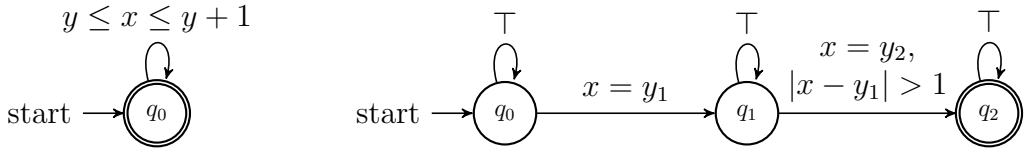
Proposition 4.14 (determinism per assignment and strong determinism)**.** If a PA A is both deterministic per assignment and strongly deterministic, it is equivalent to a symbolic automaton.

Proof. Let A be a PA that is both deterministic per assignment and strongly deterministic. Let μ be an arbitrary parameter assignment. Since A is deterministic per assignment, every word $w \in D^*$ completes a run in A_μ. Since A is strongly deterministic, no complete run of w using any other parameter assignment μ' can differ from w's run in A_μ. Therefore, $L(A) = L(A_\mu)$. □

In conclusion, any parametrized automaton not equivalent to a symbolic automaton (i. e., is parametrized in any meaningful way) cannot be strongly deterministic and deterministic per assignment at the same time. This mirrors the observations made in variable automata compared to their pattern automatons, now formally proven.

PA that are deterministic per assignment are rarely strongly deterministic, implying that we would benefit from some flexibility when handling PA. Each notion of determinism is useful in different situations, so instead of hoping that the PA we want to work on is already strongly deterministic, we should rather shift our focus on whether we could, theoretically, transform the PA into an equivalent SDPA.

We can already tell this is only possible to a limited extent, drawing on previous results. Since every SDPA can be complemented, and there are parametrized automata that cannot be complemented, obviously some parametrized automata cannot be determinized, i. e., they are not equivalent to any SDPA. It will now be shown that there are even some complementable PA that cannot be determinized. The proof will provide a counterexample of a PA that can be complemented, but not determinized.



(a) A_3, as seen in figure 3.1. (b) The complement automaton A_3^c.

Figure 4.5: A_3 and a PA A_3^c identifying the complement of $L(A_3)$, showing that A_3 can be complemented.

Theorem 4.15. Strongly deterministic PA form a strict subset of complementable PA. There are complementable PA that do not have a strongly deterministic equivalent.

Proof. Consider example A_3. $L(A_3)$ is the language of all words whose letters can be placed within an interval of length 1. A_3 uses only one parameter y, one state, which is marked as accepting, and one self-loop labeled $y \leq x \leq y+1$. If for a word w there exists a parameter value $\mu(y) \in D$ describing the upper and lower bounds of the interval, the

run terminates in an accepting state. If an unsuitable $\mu(y)$ is guessed, the run simply dies, and therefore any words lacking such a $\mu(y)$ never terminate in an accepting position.

If we complete A_3 by adding a second, non-accepting state entered upon failing this guard, then any "wrong guesses" of $\mu(y)$ will lead to a non-accepting run, making the PA non-deterministic.

In order to prove there is no SDPA describing $L(A_3)$, we prove there is no strongly deterministic PA describing its complement language $K = \{w = (w_1, \ldots, w_k) \in D^* \mid \exists 1 \leq i < j \leq k : |w_i - w_j| > 1\}$.

Consider the sequence of words $((1, \frac{1}{2}, \ldots, \frac{1}{n}, 1 + \frac{1}{n-1}))_{1<n\in\mathbb{N}}$, whose first few elements are the following:

$$\begin{array}{ll} n = 2 & (1, \frac{1}{2}, 2) \\ n = 3 & (1, \frac{1}{2}, \frac{1}{3}, 1 + \frac{1}{2}) \\ n = 4 & (1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, 1 + \frac{1}{3}). \end{array}$$

Assume that there is an SDPA A identifying K, and let A consist of k states. Each word of the sequence has to be accepted by A upon reaching the last letter, and all runs of prefixes have to terminate in non-accepting states. We will prove by induction that A cannot be deterministic by showing that all prefixes have to end in states that are pairwise distinct. This, in turn, proves A must have an unbounded number of states.

Claim: For each word $(1, \frac{1}{2}, \ldots, \frac{1}{n}, 1 + \frac{1}{n-1})$, all runs of prefixes $(1, \frac{1}{2}, \ldots, \frac{1}{i})$ where $i \leq n$ terminate in non-accepting states that are pairwise distinct, implying that a run of the word $(1, \frac{1}{2}, \ldots, \frac{1}{n}, 1 + \frac{1}{n-1})$ traverses $n + 1$ distinct states.

Base case: Let there be a configuration of parameters μ_2 such that $(1, \frac{1}{2}, 2)$ is accepted. Then (1) and $(1, \frac{1}{2})$ cannot terminate in the same state, as otherwise $(1, 2)$ would be a word terminating in an accepting state despite not being part of the language. Conclude that the automaton has at least 3 distinct states.

Induction step: Assume that the unique run of the word $(1, \frac{1}{2}, \ldots, \frac{1}{n-1}, 1 + \frac{1}{n-2})$ traverses n distinct states, using a parameter assignment μ_{n-1}. Now let μ_n be a parameter assignment such that the word $(1, \frac{1}{2}, \ldots, \frac{1}{n}, 1 + \frac{1}{n-1})$ completes a run.

Since A is strongly deterministic, for each prefix $(1, \frac{1}{2}, \ldots, \frac{1}{i})$ where $i < n$, the runs in $A_{\mu_{n-1}}$ and A_{μ_n} have to coincide. We only need to prove that the unique run of $(1, \frac{1}{2}, \ldots, \frac{1}{n})$ in A_{μ_n} does not terminate in a state that has been previously traversed.

Assume for a proof by contradiction that there is an $i < n$ such that the runs of $(1, \frac{1}{2}, \ldots, \frac{1}{i})$ and $(1, \frac{1}{2}, \ldots, \frac{1}{n})$ terminate in the same state q. Then there is a transition exiting q that leads to an accepting state upon reading the letter $1 + \frac{1}{n-1}$, so the word $(1, \frac{1}{2}, \ldots, \frac{1}{i}, 1 + \frac{1}{n-1})$ completes an accepting run in A_{μ_n}. Since $(1, \frac{1}{2}, \ldots, \frac{1}{i}, 1 + \frac{1}{n-1})$ is not part of the language K, the runs of $(1, \frac{1}{2}, \ldots, \frac{1}{i})$ and $(1, \frac{1}{2}, \ldots, \frac{1}{n})$ have to terminate in distinct states.

The final state is accepting and therefore cannot coincide with any state that has been previously traversed. Therefore, the run of the word $(1, \frac{1}{2}, \ldots, \frac{1}{n}, 1 + \frac{1}{n-1})$ traverses $n + 1$ distinct states.

A run of the word $(1, \frac{1}{2}, \ldots, \frac{1}{k}, 1 + \frac{1}{k-1})$ needs to traverse $k + 1$ distinct states, contradicting the assumption of A only having k states.

Using the closure properties of SDPA, we conclude there is no SDPA describing $L(A_3)$. □

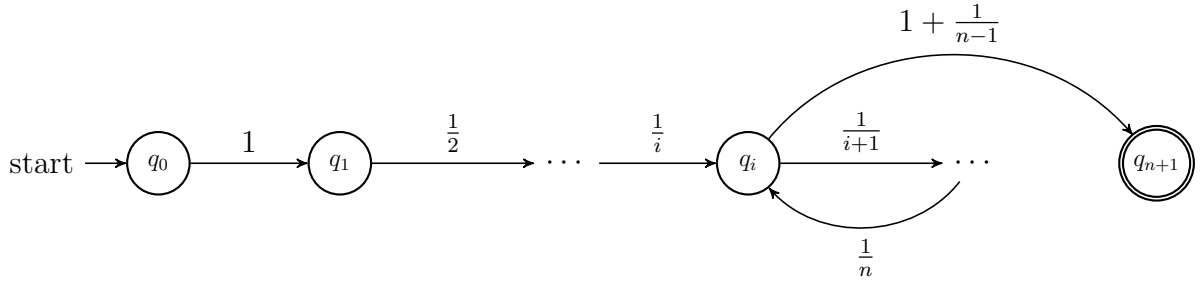


Figure 4.6: The proof, illustrated.

After establishing SDPA as a strict fragment of complementable PA, as a result determinization of PA should consist of two steps: First, the PA must be established to be determinizable in the first place. Then, an algorithm should be applied which constructs an equivalent SDPA.

Unfortunately, we would be flying too close to the sun: These two steps combined provide a means of solving the universality problem, which we know is undecidable for PA.

Theorem 4.16. At least one of the following statements holds:

1. The problem of finding out whether a PA can be determinized is undecidable.
2. There is no algorithm which can compute an equivalent SDPA for any given (determinizable) PA.

Proof. This theorem is a consequence of the undecidability of universality for general PA.

Assume that both statements are wrong. In that case, we can construct a Turing machine M that takes a PA as input and returns either an equivalent SDPA, or returns that such a SDPA does not exist. As seen in section 3.5 and specifically figure 3.8, an SDPA U such that $L(U)$ is universal can be easily constructed.

Now, M can be utilized to find out if an arbitrary PA A is universal: Run M on A. If there is no equivalent SDPA, then A cannot be universal. If M returns an equivalent SDPA S, we can easily check whether $L(S)$ is universal. This contradicts the fact that universality is undecidable for PA. □

The section will be concluded with a question that has, so far, been overlooked. Given an arbitrary PA, can we somehow confirm whether this PA is strongly deterministic?

There have been several situations already in which such a method would have been useful. The decidability results in section 4.1 rely on the user already knowing whether the PA in question are strongly deterministic. In the face of general determinization being undecidable, the ability to confirm whether the goal is reached is an important partial step in the right direction.

Strong determinism is composed of conclusiveness and completeness. We can therefore investigate both of these properties in an isolated setting, starting with conclusiveness.

Proposition 4.17. Conclusiveness of a PA is decidable.

Proof. Let $A = (M, Q, q_0, \delta, F)$ be a PA. Let $A \times A$ be the set of direct product automata of A with itself (importantly, the parameters in one copy of A need to be relabeled). In

the direct product automaton $(A \times A, F)$, let the set of accepting states be $F = \{(p, q) \in Q \times Q : p \neq q\}$. The run of a word in $(A \times A, F)$ will terminate in F if and only if there are two runs of w in A terminating in distinct states. Therefore, A is conclusive if and only if $(A \times A, F)$ is empty. □

Given a conclusive PA, it is tempting to attempt to "complete" it (*especially* given the not very promising results regarding completeness, which we will get to later). Maybe we can make it complete and therefore deterministic by simply adding a few states and transitions? Unfortunately, we have already seen a counterexample of a PA that is conclusive, yet cannot be determinized.

Corollary 4.18. Conclusive PA cannot always be determinized: There are conclusive PA that are not equivalent to any SDPA.

Proof. Consider A_3 once more: A_3 is conclusive by virtue of only consisting of one state. As shown in the proof for theorem 4.15, there is no SDPA that is equivalent to A_3. □

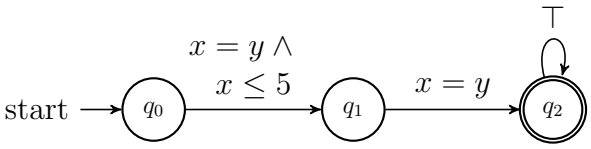
The problem with determinizing conclusive PA lies in the conflict between local and global definitions of determinism: Completeness per assignment is obtained by adding a "sink" state to which all runs are redirected that would otherwise die. Algorithmically, this is achieved by looking at each state locally, constructing a formula by negating the union of all formulae labeling exiting transitions, and using it to label a transition leading from the state to the sink state. The sink cannot be left, expressed by the only exiting transition leading straight back to the sink state and being labeled with $\top$.

Applying this to all states of a conclusive PA will, however, lead to a PA that is deterministic per assignment. As has been already shown in lemma 4.14, a PA that is deterministic per assignment is likely not strongly deterministic. Completeness per assignment is too much of an infringement on the delicate nature of strong determinism, which is defined by the interplay of different parameter assignments.

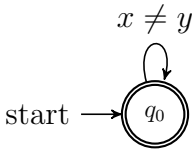
These observations smoothly translate to the problem of checking for completeness in general. A systematic, "localized" approach would check each state individually: **For every state, every letter and every parameter assignment, is there always a viable exiting transition?** It is the approach that works for FSA and SFA – and therefore overapproximates the problem, since it actually checks for the much stricter property of completeness per assignment. Complete PA are, in general, not complete per assignment: If a run of a word fails using one parameter assignment, it may still terminate when using a different parameter assignment "picking up the slack".

If the described conditions are too strict to detect completeness reliably, and we do not want to abandon the idea of a localized approach yet, maybe there is a way to relax the conditions. We do not need a word to complete a run using every parameter assignment. One parameter assignment is enough to ensure completeness. **For every state and every letter, is there always some parameter assignment permitting a viable exiting transition?**

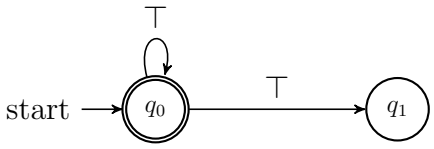
Unfortunately, this adjusted, localized approach can fail on two fronts, because it will both falsely identify complete PA as incomplete and falsely identify incomplete PA as complete.



(a) An example in which transitions are constricted by previous choices.



(b) A PA which is only complete if the domain D is infinite.



(c) A PA which may be falsely identified as incomplete, as no viable transitions exit q_1.

Figure 4.7: Examples illustrating challenges of the completeness problem.

If the PA in question is not deterministic per assignment, the method fails to recognize how a run may die on one path while terminating correctly when choosing different transitions, causing complete PAs (as seen in figure 4.7c) to be misidentified as incomplete.

On the other hand, the path previously taken may impose constraints on the current options for parameters, rendering some transitions unviable even if their guards are satisfiable locally. Therefore, an incomplete PA may be falsely identified as complete. This can be observed in figure 4.7a. The second transition, $x = y$, can be satisfied by every letter when choosing $\mu(y) = x$. However, state q_1 can only be reached by first traversing the transition labeled $x = y \wedge x \leq 5$, constricting $\mu(y)$ to values not greater than 5.

Both types of error occur because the localized approach fails to take prior decisions made during a run into account. The idea of checking each state in isolation for some desired properties should therefore be abandoned in favor of a more "globalized" approach that takes prior choices into account.

This gives rise to another issue, illustrated in figure 4.7b: It cannot be predicted *how often* a transition needs to be traversed in order to find a word which cannot complete a run. The pictured automaton accepts all words not containing a letter determined by the parameter assignment. If the domain is infinite, such a letter can always be found because each word only contains a finite number of (distinct) letters, and therefore the PA is universal. If, however, D is finite, e. g., $D = \{1, 2, \ldots, n\}$, the word $12 \ldots n$ will never complete a run. The length of a shortest word that does not complete a run is not related to the number of states or number of parameters.

Since a transition may need to be traversed multiple times, it is not sufficient to iterate through all non-looping paths in the automaton and perform satisfiability checks. Nor can we put an upper bound on the length of the words that need to be tested. However, these considerations help get a better estimate of the computability of the problem.

Lemma 4.19. The set of all incomplete PA is semidecidable.

Proof. We prove the statement by constructing an algorithm which, given an input PA A, halts when it finds a word that does not complete a run in A. If no such word exists, the algorithm runs forever.

For a $n \in \mathbb{N}$, let there be n distinct variables $x_1, x_2, \ldots, x_n$ that do not occur in formulae of A. Let $r = ((q_0, r_1, q_1), (q_1, r_2, q_2), \ldots, (q_{n-1}, r_n, q_n))$ be a path of length n.

Each formula r_i may contain the free variable x, as well as k parameters $\{y_1, y_2, \ldots, y_k\}$. We transform each formula r_i by replacing x with the variable x_i and denote the resulting formula $r_i^{x_i}$. Therefore, a word $w = w_1 w_2 \ldots w_n$ can complete the run r if and only if the formula

$$\bigwedge_{i=1}^{n} r_i^{x_i}(w_i)$$

is satisfiable, with $r_i^{x_i}(w_i)$ implying that x_i in r_i is mapped to w_i.

Since A contains a finite number of transitions, we can enumerate all distinct paths of length n. Let $\Gamma_n(A)$ be the finite set of all distinct paths of length n. To check if all words of length n complete a run in A, we need to verify whether the formula

$$\forall w_1, w_2, \ldots, w_n. \bigvee_{r \in \Gamma_n(A)} \bigwedge_{i=1}^{n} r_i^{x_i}(w_i)$$

is satisfiable. If it is not, we have found proof that A is not complete.

Iterating through all $n \in \mathbb{N}$, we have obtained the desired algorithm.

□

The completeness problem may be closely related to the universality problem. A PA is complete if and only if the PA resulting from marking all of its states as accepting is universal. The conditions are not right to use the universality problem in a reduction to prove that confirming completeness is undecidable. Likewise, this approach likely won't lead to an applicable method for identifying complete PA, even if universality only needs to be shown for a very specific subtype of PA.

Conjecture 4.20. Deciding whether a PA is complete is undecidable. This also means deciding whether a PA is strongly deterministic is undecidable.

The observations made in this section cement the impression of strong determinism being a very unreliable criterion when aiming to complement PA. We have seen that SDPA have pleasant closure properties and the most important decision problems are decidable, however these results rely on the user already knowing whether the PA in question is strongly deterministic. When attempting to transform an arbitrary PA into an SDPA, we quickly run into the undecidability barrier. There are also complementable PA that are not equivalent to any SDPA.

Therefore, SDPA are most useful when they are already given and do not have to be constructed manually, since then we can fully benefit from the very efficient complementation algorithm described in this section. If the starting point is not an SDPA, however, we should turn to different methods that will be explored in chapter 6.

Chapter 5

Parameter Management

In previous sections, investigations of the properties of the parameter assignment itself have been very limited. The relationship between parameter assignments and words has been unidirectional: For a given word which is part of a PA's language, there exists a parameter assignment such that the word completes a run. However, this relationship also works in the other direction. For a given parameter assignment, which words complete runs, and which runs terminate in accepting states?

We also wish to be able to make more nuanced statements about the parameter assignment instead of being constrained to existence results, which may open the way for new approaches with respect to the complementation problem. Compared to PA, VA have much tighter "control" over their parameters, as all parameters relevant for the run of a word need to correspond directly to letters of the word. In a PA, the parameter values do not have to correspond to letters of the word directly, and often there is more than one parameter assignment such that a word completes an accepting run.

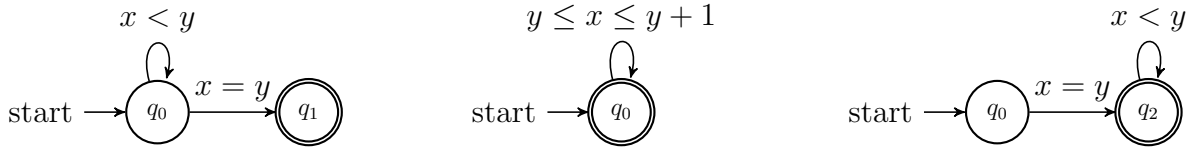
There is much to explore. Parameter assignments can be categorized. For example, in an SDPA, the set of all parameter assignments Θ splits in two subsets for every word w: The parameter assignments for which w does not complete a run, and those for which it does. In the latter subset, all runs have to be identical.

This section seeks to establish concepts and vocabulary allowing for a more fine-grained description of the parameter assignment and its properties. We will contend ourselves with a cursory investigation.

5.1 Strict Parameters

Example 5.1. Consider A_2 and A_3 in figure 5.1. As seen before, A_2 is deterministic per assignment but not strongly deterministic. A_3 is neither, as words outside of $L(A_3)$ will never complete a run.

Observe that for every $w \in L(A_2)$, there exists only one parameter assignment μ such that $w \in L((A_2)_\mu)$. Namely, $\mu(y)$ has to coincide with the last letter of w. The choices in A_3 are less constrained, as for many words there is more than one parameter assignment using which the word completes a run. Let $w \in L(A_3)$ such that the smallest letter of w is z and the largest letter of w is $z + \varepsilon$ where $\varepsilon < 1$. Then every parameter assignment μ where $\mu(y) \in [z + \varepsilon - 1, z]$ will lead to an accepting run. The parameters of A_2 are "stricter" than those of A_3, in the sense that the "range" of parameters accepting a word



(a) A_2 (b) A_3 (c) A_2^R

Figure 5.1: Three automata that display different levels of "control" over their parameter choices.

is smaller.

This observation is limited to accepting runs, but can be generalized to all runs. In A_2, the run of a word $w \in L(A_2)$ can terminate in q_0 or q_1, depending on the parameter assignment. Up until the last transition, the parameter y does not need to be assigned a fixed value as long as $\mu(y)$ is greater than all upcoming letters. The run's final state is decided upon the last transition, depending on whether $\mu(y)$ coincides with the last letter of w or not. Up until that point, we do not know whether the run will terminate in q_0 or q_1. In contrast, A_2^R will "force" the assignment of y to the first letter of the input word. This kind of pressure on the parameters appears to be crucial for strong determinism. After leaving the initial state, A_2^R will act like a symbolic automaton on the remaining suffix of the word because $\mu(y)$ has been predetermined. In contrast, the transition $(q_0, x < y, q_0)$ in A_2 is "too loose", permitting every word to complete a run terminating in q_0 as long as $\mu(y)$ is larger that the largest letter of the word.

The latter notion can be formalized.

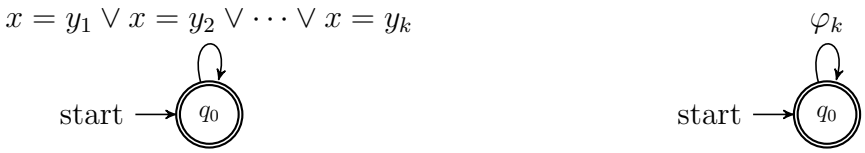
Definition 5.2. A PA A is strict (or has strict parameters) if for every word $w \in D^* \setminus \{\varepsilon\}$, there exists at most one parameter assignment μ such that $|p_\mu(w)| > 0$, i. e., w completes a run in A.

A PA A is strict-when-accepting if for every word $w \in L(A) \setminus \{\varepsilon\}$, there exists exactly one parameter assignment μ such that $|p_\mu^a(w)| > 0$, i. e., w completes an accepting run in A.

The empty word ε is excluded from the definition because it trivially completes the "empty run" using every parameter assignment.

Example 5.3. The automaton A_3 is neither strict nor strict-when-accepting, A_2 is strict-when-accepting but not strict, and A_2^R is both strict and strict-when-accepting.

Example 5.4 (strictness-when-accepting in PA with more than one variable)**.** For a fixed number k, let L^k be the language of all words composed of at most k distinct letters, and A_4 be a PA identifying L^k.



(a) A PA for L^k, A_4. (b) A_4 with alternative notation.

Figure 5.2: A PA A_4.

We will attempt to transform A_4 into a PA that is strict-when-accepting.

The PA A_4 has k parameters at its disposal. If a word w is composed of $l \leq k$ distinct letters $\{a_1, a_2, \ldots, a_l\}$, then it is accepted using any parameter assignment μ whose image contains all of $\{a_1, a_2, \ldots, a_l\}$. However, A_4 is not strict-when-accepting, since any permutation of μ will also accept w. We denote φ_i as a shorthand for the formula $x = y_1 \vee x = y_2 \vee x = y_i$.

A first step towards strictness-when-accepting could be to demand that $\mu(y_1)$ correspond to the first distinct letter in w, $\mu(y_2)$ to the second distinct letter (i. e., the first letter that is not equal to $\mu(y_1)$) and so on.

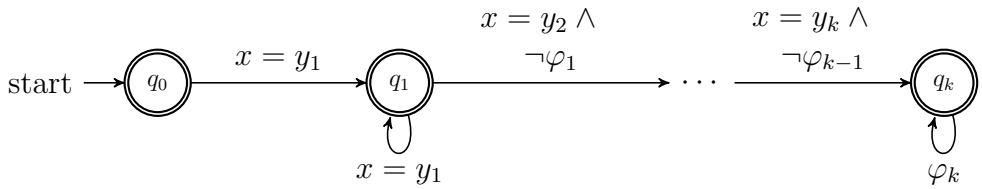


Figure 5.3: A PA for L^k after modifications.

Observe that this step has led to a k-fold increase in the number of states. Also, the PA is not strict-per-assignment. Words with fewer than k distinct letters never enter state q_k, so $\mu(y_k)$ never needs to be defined and can be assigned arbitrarily. This can be fixed by forcing the first $k-l$ parameters to be assigned the first letter of a word with l distinct letters.

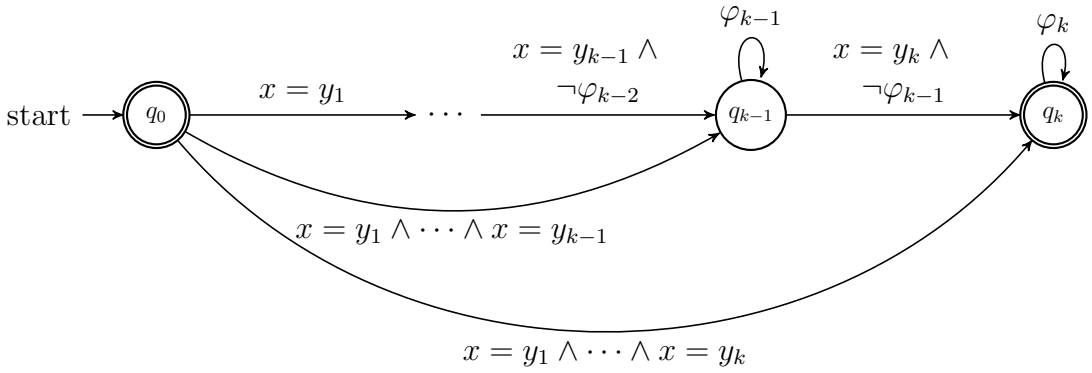


Figure 5.4: A PA for L^k that is strict-when-accepting.

Note that in the final version, only q_0 and q_k are accepting states. The run of a word with $l < k$ distinct letters can only "reach" q_k if y_1 through y_{k-l} are assigned the first letter.

5.2 Strictness and SDPA

Strictness is a useful property, since it can help out in a situation that has previously appeared to be a dead end. In general, conclusive PA cannot be transformed into SDPA. However, this changes if the PA has strict parameters.

Theorem 5.5. Let A be a conclusive PA with strict parameters such that every word of length 1 completes a run. Then there exists an SDPA equivalent to A.

Proof. Let $A = (M, Q, q_0, \delta, F)$ be a conclusive PA with strict parameters, and w a word that does not have a terminating run in A. If such a w does not exist, then A is already complete and the proof is done. Also w cannot be the prefix of a word that has a complete run.

Let v be the longest prefix of w such that v has a complete run in A. Let μ be the corresponding parameter assignment, and q be the state in which the run of v terminates. Since A is conclusive, q is well-defined. Then there is a letter a such that va is a prefix of w and va does not complete a run in A, since we assumed v to be the largest prefix of w that completes a run.

Then create a (non-accepting) dumpster state $p \notin Q$, a transition $(p, \top, p)$ and a transition (q, φ, p) where φ is the negation of the conjunction of all other guards exiting q, namely $\varphi = \neg \bigvee_{\psi \in S} \psi$ for $S = \{\psi \mid (q, \psi, s) \in \delta \text{ for some } s \in Q\}$. The state q is now "locally complete", in the sense that for every letter and every parameter assignment, there has to be a viable exiting transition.

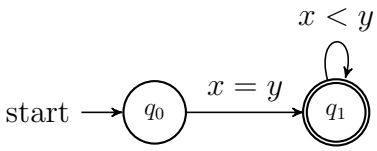
We also need to avoid q being the initial state, since the empty word is exempt from needing to be accepted by a single parameter assignment. Therefore, if $q = q_0$, let there be a new state $q_0' \notin Q \cup \{p\}$ that is accepting if and only if q_0 is accepting, and a set of transitions $T = \{(q_0', \varphi, s) \mid (q_0, \varphi, s) \in \delta\}$.

Let

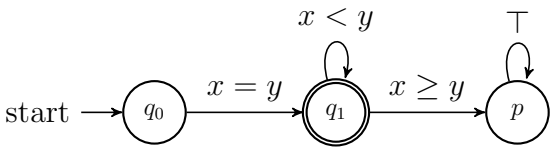
- $A' = (M, Q \cup \{p\}, q_0, \delta \cup \{(p, \top, p), (q, \varphi, p)\}, F)$ if $q \neq q_0$,
- $A' = (M, Q \cup \{p, q_0'\}, q_0', \delta \cup \{(p, \top, p), (q, \varphi, p)\} \cup T, F)$ if $q = q_0$ and $q \notin F$, or
- $A' = (M, Q \cup \{p, q_0'\}, q_0', \delta \cup \{(p, \top, p), (q, \varphi, p)\} \cup T, F \cup \{q_0'\})$ if $q = q_0$ and $q \in F$.

The resulting PA A' is still conclusive, which needs to be proven. Assume by contradiction that there is a word $u \in D^*$ such that u completes two distinct runs in A'. A' is obviously conclusive per assignment, since A was conclusive per assignment and, for every fixed parameter assignment, the recently added transition from q to p is mutually exclusive with any other transition leaving q. The state q_0', if added, does not violate conclusiveness because q_0 does not violate conclusiveness. Therefore, the two runs must have occurred using two different parameter assignments μ_1, μ_2. The original automaton A was conclusive, so at least one of the runs had to traverse one of the newly added transitions and terminated in p. Every transition before that point occurred in A and has to coincide for both parameter assignments. Thus, there is a prefix of u whose runs lie entirely in A, and since A was assumed to have strict parameters, the prefix has to be empty. This contradicts the fact that q is not the initial state of A'. Therefore, such a u cannot exist. The same argument preserves strictness in A'.

This procedure can be applied iteratively. Note that a new initial state q_0' only needs to be added once (although not necessarily in the first iteration), because words of length 1 always complete a run and there are no loops back to q_0'. We select a new word w_2 that does not complete a run in A'. Its longest prefix v_2 that completes a run cannot terminate in q, since q was "locally complete". Let the run of v_2 terminate in some state q_2. Since p has already been added in the first step, we only need to add the transition (q_2, φ_2, p) where $\varphi_2 = \neg \bigvee_{\psi \in S} \psi$ for $S = \{\psi \mid (q_2, \psi, s) \in \delta \text{ for some } s \in Q\}$. Rinse and repeat.



(a) A_2^R, a PA that is conclusive and incomplete.



(b) D_2, a PA that is equivalent to A_2^R and strongly deterministic.

Figure 5.5: An example of a strict, conclusive PA that has been transformed into an SDPA.

Since there is a finite number of states and the number of states that can be modified decreases after each step, this method terminates. □

The challenge of this method lies in finding words which do not complete a run, related to the completeness problem. Fortunately, this issue can be entirely circumvented: Every original state of A can be "locally completed" without causing the resulting PA to become inconclusive. Let A' be the automaton obtained by first making A deterministic per assignment and then adding the new initial state q_0' as described in the proof. Every run of every word can only exit q_0' using one unique parameter assignment, and the remainder of the run occurs in a PA that is conclusive per assignment. Therefore, each run in the SDPA A' splits into two phases: An "assignment phase" in which the parameter values needed for a complete run are determined, and a "stable phase" afterwards where the run operates on a subsection of A' that is deterministic per assignment.

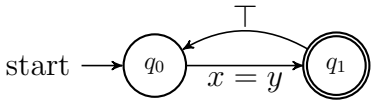


Figure 5.6: A PA A fulfilling all conditions of theorem 5.5.

Example 5.6. The PA A seen in figure 5.6 accepts all words of odd length whose first letter appears in every odd position. Every word using every parameter assignment has to take the path q_0 - q_1 - q_0 - q_1 - ... assuming the run does not die, so A is conclusive. Every word $w \in D$ of length 1 is accepted by A_μ where $\mu(y) = w$. The PA is strict, since a word $w_1 w_2 \dots w_n$ only completes a run when $\mu(y) = w_1$.

All words of length 0, 1 or 2 complete a run. A word that does not complete a run is $(1, 2, 2)$. The longest prefix of w that completes a run is $(1, 2)$ and the run terminates in q_0. Therefore, both a state p and a state q_0' need to be added.

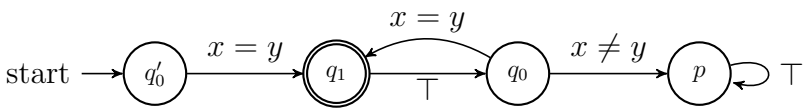


Figure 5.7: A' after one step of completing. For improved legibility, the positions of q_0 and q_1 have been swapped.

The resulting PA A' is complete, therefore we are finished after one step.

Note that SDPA are not necessarily strict. Two examples can be seen in figure 5.8: Both automata are strongly deterministic, but are not strict for different reasons. In the first automaton, the parameter y_2 is not "needed" for words of length 1 and can therefore be assigned arbitrarily. In the second, there is more than one possible choice for y. Strong determinism is retained because in all subsequent transitions that use y, all potential choices for $\mu(y)$ have equivalent properties.

Both examples can be transformed into PA that are strict, which we will look into next.

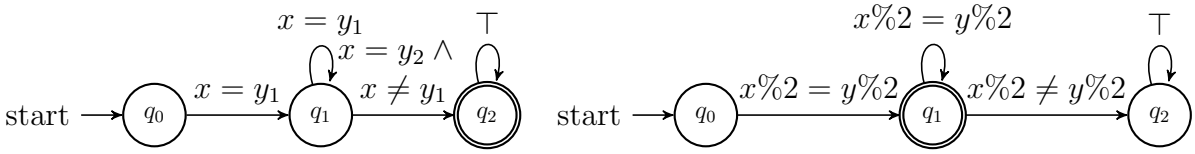


Figure 5.8: SDPA which are not strict.

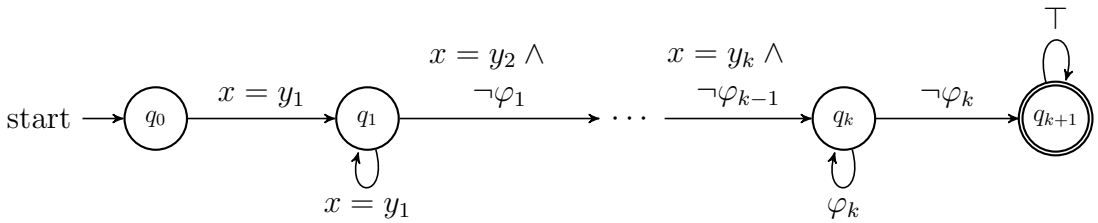


Figure 5.9: A strongly deterministic complement automaton for A_4.

Remark 5.7. Remember the automaton A_4 from example 5.4, which is complementable: We can obtain a complement PA for L^k by starting with the PA from figure 5.3, making all states non-accepting and then adding a new accepting state p and the transitions $(q_k, \neg\varphi_k, p)$ and $(p, \top, p)$. State q_k is only entered when k distinct letters have been encountered, allowing us to transition to an accepting state when a new, previously unknown letter appears. The resulting automaton can be seen in figure 5.9.

Interestingly, there are parallels to the observations made after theorem 5.5. The theorem states that if a PA A is conclusive, has strict parameters and every word of length 1 completes a run, it can be transformed into an SDPA A' by making the original PA deterministic per assignment and adding a new initial state q'_0, which is a copy of the old initial state and all of its exiting transitions.

We have observed that each run through A' splits into two "phases". In the first phase, which we dubbed the "assignment phase", the parameters needed for a complete run are determined. In the second phase, or the "stable phase", the remainder of the word runs through a subsection of A' that is deterministic per assignment. The two phases are directly reflected by the layout of A': The second phase, by definition, occurs in the section of A' that was obtained by making the original PA A deterministic per assignment. The first phase consists of a single transition exiting the initial state q'_0 (which can never be reentered) and entering one of the states of the section used in the second phase. Therefore, A' can be divided into two separate subsections which directly correspond to the two phases. It is therefore more accurate to speak of an "assignment section" and a "stable section".

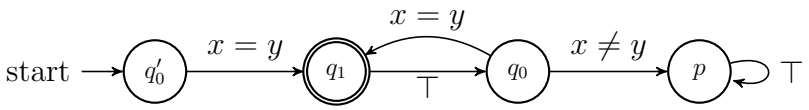


Figure 5.10: An example of a PA A', as seen in example 5.6.

In the example depicted in figure 5.10, the assignment section consists of the state q_0' and its exiting transition $(q_0', x = y, q_1)$. The remainder of A' is the stable section.

This callback to the earlier theorem is relevant because we can observe the same division into two sections in A_4^c. In stark contrast to A', most of A_4^c is part of the assignment section. The stable section begins with state q_k: Upon entering q_k, all parameter values have been determined, and the section of the automaton that is reachable from q_k is deterministic per assignment. The preceding states q_0 through q_{k-1} and all transitions exiting those states serve to identify suitable parameter assignments.

Two important conclusions can be drawn from observing A_4^c: The split into an assignment section and a stable section can also be observed in SDPA that are not strict, and the assignment section can have arbitrary size.

It remains an open question whether every SDPA has this structure.

5.3 Are Strict Parameters Obtainable?

We have seen how strictness can help out when determinizing PA, yet the conditions which need to be met seem very oppressive. In the following subsection, the prevalence of both strictness and strictness-when-accepting will be examined.

Every strict PA is strict-when-accepting, but not every PA which is strict-when-accepting is strict. The question remains whether every strict-when-accepting PA can be transformed into a strict PA, and whether every arbitrary PA can be transformed into a strict or strict-when-accepting PA. The subset relations in question are depicted in figure 5.11.

Theorem 5.8. There are PA that are not equivalent to a strict PA.

Proof. Let $L_2 = L(A_2)$ be the language of all words in which the last letter is strictly largest. Assume that there is a strict PA A such that $L(A) = L_2$.

Since A is strict, for every $w, v \in L(A)$ where v is a prefix of w it holds that w and v must be accepted by the same parameter assignment. We will construct a sequence of words using this property to prove A cannot have a finite number of states.

Consider the sequence of words $((1, 2, \ldots, n))_{n\in\mathbb{N}}$, whose first few elements are the following:

$$\begin{aligned} n &= 1 \quad (1) \\ n &= 2 \quad (1, 2) \\ n &= 3 \quad (1, 2, 3). \end{aligned}$$

Since $(1) \in L_2$ and $(1, 2) \in L_2$ and (1) is a prefix of $(1, 2)$, there exists a unique $\mu \in \Theta$ such that both (1) and $(1, 2)$ complete accepting runs in A_μ. While A is strict, we cannot assume it is conclusive, so there may well be more than one possible run which (1) and $(1, 2)$ can complete in A_μ. This complicates the proof.

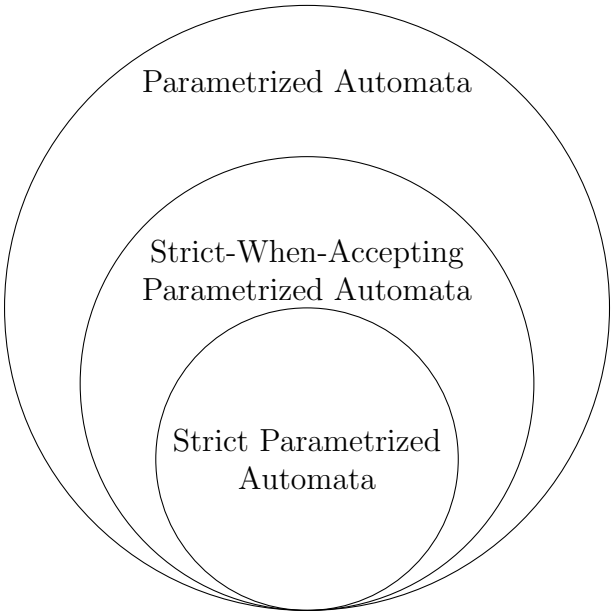


Figure 5.11: Relations between the different classes of parametrized automata. The question remains whether the represented inclusion relations are strict up to equivalence of automata: Is every PA equivalent to a PA that is strict or strict-when-accepting, and are there PA that are strict-when-accepting but not equivalent to a strict PA?

However, it can be deduced in the accepting run of $(1, 2)$, the last transition taken cannot be a self-loop. Let this transition be labeled (q_1', φ_1, q_2), where q_2 is an accepting state, q_1' is a terminating state of a run of (1) and φ permits the letter 2. If $q_1' = q_2$, there would be a self-loop from and to the final, accepting state that can be traversed by the letter 2, causing the word $(1, 2, 2)$ to be falsely accepted. Therefore, there are runs of (1) and $(1, 2)$ terminating in distinct states, meaning that A consists of at least two states. It does not matter whether, in this particular case, the run of (1) terminates in an accepting state or not.

This principle holds for the accepting run of $(1, 2, 3)$ as well: Let (q_2', φ_2, q_3) be its last transition, where q_3 is an accepting state, φ_2 permits the letter 3 and q_2' is a terminating state for a run of $(1, 2)$. The same line of reasoning as above enforces $q_2' \neq q_3$. Yet now, we can also deduce $q_2' \neq q_1'$, because q_2' is a terminating state for a run of $(1, 2)$ and there is a transition from q_1' to the accepting state q_2 which permits the letter 2, leading to a falsely accepting run of the word $(1, 2, 2)$. Finally, $q_1' \neq q_3$ holds because otherwise, the word $(1, 2, 3, 2)$ would have a falsely accepting run. The automaton A therefore needs to have at least 3 states, because the states q_1', q_2' and q_3 are all pairwise distinct.

To generalize this observation to all n, let q_{n-1}' be the second-to-last state of an accepting run of $(1, 2, \ldots, n)$. The state q_{n-1}' cannot be equal to q_{i-1}', the second-to-last state of an accepting run of any other prefix $(1, 2, \ldots, i)$, because otherwise the word $(1, 2, \ldots, n-1, i)$ would have a falsely accepting run.

Each word $(1, 2, \ldots, n)$ has to be accepted by a PA with at least n states, proving there is no upper limit on the number of states of A. Therefore, A cannot exist. □

Since A_2 is strict-when-accepting, this proves strict PA form a non-trivial subset of both PA and strict-when-accepting PA.

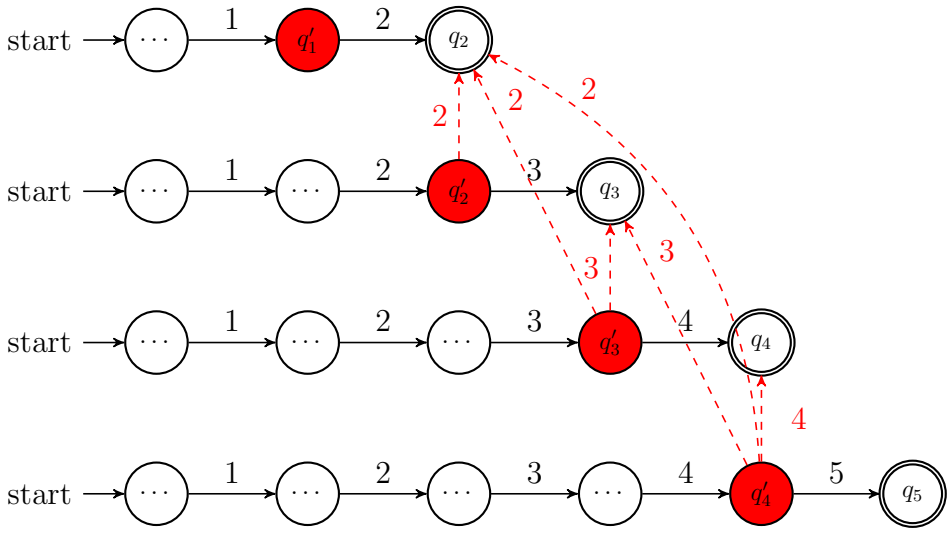
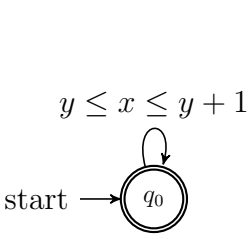


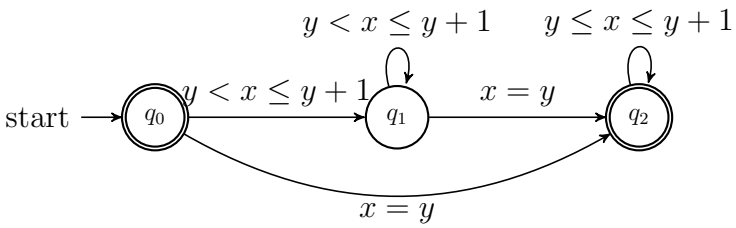
Figure 5.12: The proof, illustrated by a "flattened" depiction of the different runs. Each row represents the accepting run of a word $(1, 2, \ldots, n)$. Different labels may refer to the same state, even within a run. States which are proven to be distinct are marked in red, with dashed lines represent transitions that lead to falsely positive, accepting runs.

Next, the relationship between general PA and PA that are strict-when-accepting should be investigated. For this purpose, we will use a few examples to gauge the level of difficulty in transforming arbitrary PA into PA with strict parameters.

Example 5.9. The PA A_3 is neither strict nor strict-when-accepting, but we can attempt to transform A_3 into a PA that is strict-when-accepting. As observed earlier, for every word whose maximum distance of letters is smaller than 1, there is a range of parameter assignments such that the word is accepted. In order to make the PA strict-when-accepting, we can attempt to restrict this range to one value. For example, a PA can be constructed which accepts the words of $L(A_3)$ if and only if y assumes the value of the smallest letter.



(a) A_3 from figure 3.1.



(b) A PA equivalent to A_3 that is strict-when-accepting.

Figure 5.13: Example 5.9.

Every word except for the empty word can only be accepted by state q_2, which is only entered if the current letter equals $\mu(y)$. If there is a letter $x < \mu(y)$, the run dies, ensuring $\mu(y)$ has to be the minimum. If $\mu(y)$ is a lower bound of the input letters but not a minimum, the run terminates in state q_1. Therefore, any word $w \in L(A_3) \setminus \{\varepsilon\}$ is only accepted by the single parameter assignment $\mu(y) = min\{w_i \mid w_i \text{ is a letter of } w\}$.

So far, all of the considered examples can be transformed into PA that are strict-when-accepting.

We have reason to believe that not every PA can be transformed into an equivalent PA which is strict-when-accepting. Consider L_1, the language of of all unsorted words introduced in theorem 3.4. The automaton A_1 in figure 3.1 is neither strict nor strict-when-accepting, since runs of words with multiple instances of disorder (i. e., multiple indices $i < n$ such that $w_i > w_{i+1}$ for a word $w = w_1 w_2 \dots w_n \in D^*$) have multiple opportunities to transition to the accepting state. Intuitively, a PA A_1' for L_1 that is strict-when-accepting would have to reliably pick out the same instance of disorder in every accepting run. This means that A_1' needs to keep track of previous and/or upcoming instances of disorder. If, for example, A_1' were to always transition to an accepting state upon the first instance of disorder, it would be able to identify prefixes which are entirely sorted. As already shown, a PA accepting exactly all sorted words does not exist. We will not provide a formal proof.

Conjecture 5.10. There are PA not equivalent to a PA which is strict-when-accepting.

Without digging too deep, we can formalize a few final thoughts on this section. For instance, it appears that non-complementability of L_1 by a PA is crucial for the proof that L_1 cannot be identified by a PA that is strict-when-accepting. Both a PA for the complement of L_1 and a PA for L_1 which is strict-when-accepting appear to need "too many comparisons between letters" for a PA to handle. Moreover, all examples of complementable PA that have been studied so far can be made strict-when-accepting. We therefore posit there is a connection.

Conjecture 5.11. Every complementable PA is equivalent to a PA that is strict-when-accepting.

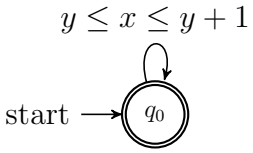
Conjecture 5.12. Every PA that is strict-when-accepting can be complemented.

Since this subject has the potential to eat a lot of time, we will leave further investigations to future researchers. As a first step, the statements could be proven for variable automata: The concept of strictness can be transferred to VA, which simultaneaously permit tighter control of the parameters.

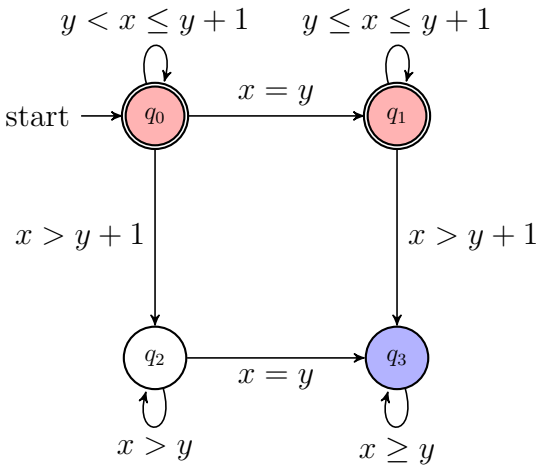
The following chapter will return to the task of finding subclasses of PA which are easy to complement.

Chapter 6

Complementable Normal Form



(a) The PA A_3 first seen in figure 3.1.



(b) A PA for L_3 in complementable normal form. State q_3 identifies the complement language.

Figure 6.1: Two automata for the language L_3 of words whose letters fall within an interval of length 1.

6.1 Idea and General Considerations

In an SDPA $S = (M, Q, q_0, \delta, F)$, the set of states splits into two disjoint subsets $Q = F \cup C$ such that all words whose runs terminate in F are part of the language $L(S)$, and all words whose runs terminate in C are part of the complement $L(S)^c$. Our new approach is to relax this notion, and introduce a new type of parametrized automaton in which the set of states Q splits into three disjoint subsets $Q = F \cup C \cup W$. A word is part of the language if it has a run terminating in F, or part of the complement if it has a run terminating in C. If a run terminates in W, we do not obtain any information about whether a word is part of the language or its complement.

A different approach to the same idea is the following: Consider a PA A and a word $w \in L(A)$. A run of w in A only terminates in an accepting state if a "suitable" parameter assignment has been chosen. If the parameter assignment does not permit the word to

complete a run in an accepting state, two things may happen: Either the word does not complete a run at all, or the run terminates in a non-accepting state q.

In the latter case, this means the state q is somewhat ambiguous. If a run terminates in q using a parameter assignment μ, we cannot tell whether the corresponding word is actually not in the language or whether μ was merely an unsuitable parameter assignment. We cannot construct a PA for the complement by turning all non-accepting states into accepting states, because then q would falsely accept words that are not part of the complement.

We are therefore looking for a way to assess the "suitability" of a parameter assignment for a specific word. Ideally, this method should be integrated into the PA in question by creating a subset of states that can only be reached by a word if the parameters are deemed "suitable". That way, if the parameters are "suitable" and the run still terminates in a non-accepting state, we can be sure the word is not part of the language.

Both approaches converge beautifully: We are looking for a PA $A = (M, Q, q_0, \delta, F)$ with a partition $Q = F \cup C \cup W$ of the set of states such that a run of a word can only terminate in the states in F (accepting $L(A)$) and C (accepting $L(A)^c$) if the parameter assignment has been deemed suitable. If the parameter assignment is not suitable, the run will terminate in W. We say such a PA is in complementable normal form.

Definition 6.1 (complementable normal form)**.** A complementable PA $A = (M, Q, q_0, \delta, F)$ is in complementable normal form (called a CFPA) if there is a subset $F_c \subseteq Q \setminus F$ such that the automaton $C = (M, Q, q_0, \delta, F_c)$ identifies the complement of A, i. e., $L(C) = L(A)^c$. States that are not in F or F_c are called *weak states*.

Example 6.2. Every SDPA is in complementable normal form. In this case, every non-accepting state is part of F_c. There are two differences between SDPA and CFPA: In an SDPA, $F \cup F_c = Q$, while a CFPA may contain additional states that do not belong to either set. Additionally, while both types of automaton are complete, a CFPA does not have to be conclusive or even deterministic per assignment.

After having locked down a formal definition of CFPA, their properties should now be investigated.

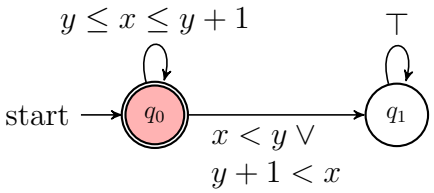
While the inclusion of weak states seems like a minor adjustment, it provides a crucial advantage of CFPA over SDPA. Theorem 4.15 shows that not every complementable PA can be determinized. But as it turns out, every complementable PA can, in fact, be brought into complementable normal form.

Theorem 6.3. For every complementable PA, there is an equivalent CFPA.

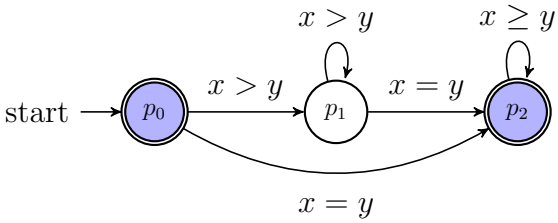
Proof. Let $A = (M, Q, q_0, \delta, F)$ be a PA and $A^c = (M, P, p_0, \delta_c, F_c)$ be the complement automaton of A. Assume without loss of generality that $Q \cap P = \varnothing$. Then A and A^c can be linked via an ε-transition similar to the construction used for the union of two automata. Let s be a new state, such that $s \notin Q$ and $s \notin P$. The PA $A' = (M, Q \cup P \cup \{s\}, s, \delta \cup \delta_c \cup \{(s, \varepsilon, q_0), (s, \varepsilon, q_0)\}, F)$ is equivalent to A and $(M, Q \cup P \cup \{s\}, s, \delta \cup \delta_c \cup \{(s, \varepsilon, q_0), (s, \varepsilon, q_0)\}, F_c)$ is equivalent to A^c. Therefore, A' is a CFPA. □

The theorem provides an upper limit on the size of a CFPA: Let n be the size of Q and m be the size of P. Then there exists a CFPA equivalent to A with no more than $n + m + 1$ states.

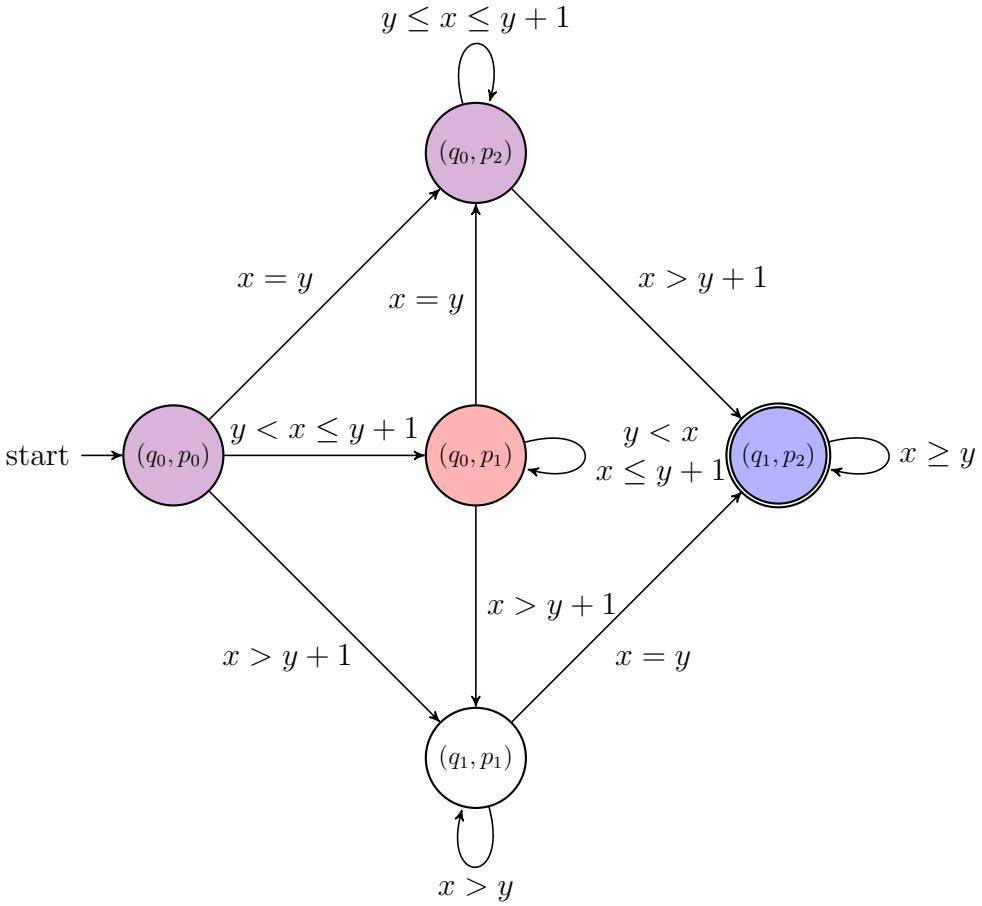
6.2 Construction of Complementable Normal Form



(a) A PA for L_3 that is deterministic per assignment, named $\langle A_3 \rangle$.



(b) A suitable Skolem automaton B.



(c) A synchronized product of both automata, which accepts the complement of L_3.

Figure 6.2: Construction of a CFPA that is equivalent to the automata in figure 6.1.

It is likely hard to prove whether a PA is in complementable normal form. On the one hand, this would easily allow to identify whether a PA is strongly deterministic, since an SDPA is a conclusive CFPA such that $F \cup F_c = Q$. On the other hand, a CFPA is complete by definition, and checking for completeness is another open problem.

Conjecture 6.4. Deciding whether a PA is in complementable normal form is undecidable.

However, there are promising results regarding the construction of a CFPA. As described in the introduction to this chapter, we are looking for a way to determine the

"suitability" of parameter assignments. Consider the PA in figure 6.2a. The run of any word $w \in L(\langle A_3 \rangle)$ will terminate in an accepting state if $\mu(y)$ equals the smallest letter in w. We will exploit the contrapositive of this statement: If the run of a word v terminates in a non-accepting state and $\mu(y)$ equals the smallest letter of v, then v cannot be a part of $L(\langle A_3 \rangle)$.

In order to formalize and then generalize this ansatz, we will introduce two new constructions. One are *Skolem automata*: Universal PA which accept any word, but only permit accepting runs if the parameter assignment fulfills certain conditions. For example, the PA B in figure 6.2b only accepts a word w if the parameter y is assigned the smallest letter of w.

The second is the *synchronized product*, which works similarly to the direct product except that it allows parameters to be assigned *dependently*. A synchronized product of $\langle A_3 \rangle$ and B only uses one parameter y, which means that the constrictions imposed on μ by B will influence a word's run in $\langle A_3 \rangle$. We are now able to identify words that complete a run in a non-accepting state in $\langle A_3 \rangle$ while $\mu(y)$ equals the smallest letter of the word, because these words will have a run terminating in the state (q_1, p_2) of the synchronized product.

In figure 6.2c, all product states (q, p) where both q and p are accepting states in their respective parent automata are marked violet. These states are accepting states for the language L_3. The product state (q_1, p_2) is marked blue: The state q_1 in $\langle A_3 \rangle$ is non-accepting while p_2 is accepting in B. The product state accepts exactly the language L_3^c. Product state (q_0, p_1) exists because some words $w \in L_3$ may be accepted by $\langle A_3 \rangle$ even if $\mu(y)$ doesn't equal their smallest letter. It is composed of an accepting state of $\langle A_3 \rangle$ and a non-accepting state of B, and marked in red. The PA can accept both L_3 and its complement and is therefore a CFPA.

Note that this was only a rough, example-driven outline of the idea behind constructing CFPA. Next, synchronized products and Skolem automata will be defined formally and illustrated by more examples before we move to combine both concepts into a method for obtaining CFPA.

Definition 6.5 (synchronized product)**.** Let there be two PA $A = (M, Q, q_0, \delta_A, F_A)$ and $B = (M, P, p_0, \delta_B, F_B)$ whose parameter sets may intersect, i. e., $Y_A \cap Y_B \neq \varnothing$. Then, analogously to the direct product, define the *set of synchronized product automata* $A \otimes B = \{(M, Q \times P, (q_0, p_0), \delta, F) \mid F \subseteq Q \times P\}$ such that $\delta = \{((q, p), \varphi_1 \wedge \varphi_2, (q', p')) \mid (q, \varphi_1, q') \in \delta_A, (p, \varphi_2, p') \in \delta_B\}$. The parameter set of $A \otimes B$ is $Y_A \cup Y_B$.

A direct product has been useful whenever we wanted to observe the terminating states of a word in two or more *independent* situations. For example, when computing the intersection, a direct product answers the question: "Can w terminate in accepting states of both parent automata?". A direct product also tells whether a PA A is conclusive, by answering the question: "Can w terminate in two distinct states of A?". Since we do not want the word's run in one automaton to be influenced by the word's run in the other automaton, it is crucial that the parameter sets of both PA be disjoint. We do not care about the parameters themselves, which only have to be modified at all if they threaten to interfere with the intended purpose of the direct product, like a malfunctioning tool.

A synchronized product has a very different relationship with the parameters of the parent automata and answers a very different set of questions. By allowing the parameter

sets to intersect or even coincide, the parent automata can no longer operate independently. The parameter choices in one parent automaton will influence the viability of transitions in the other parent automaton. As such, we can use synchronized products to impose additional constraints on transitions and parameters. In the context of CFPA, synchronized products will answer the question: "Can w terminate in an accepting state of one parent automaton, while parameter choices are restricted by the other parent automaton?".

Example 6.6. Consider the following PA:

(a) A universal SDPA E_1.

(b) A universal SDPA E_2.

Figure 6.3: Examples of universal SDPA.

Both E_1 and E_2 are universal SDPA. E_1 accepts words whose first letter corresponds to y, and E_2 accepts words whenever y is an upper bound for the letters in the word.

Now, compare the synchronized and the direct product of E_1 and E_2. In this example, choose F to be the set of product states where both original states are accepting.

(a) $(E_1 \times E_2, F)$, with the parameter of E_2 relabeled to y'.

(b) A synchronized product automaton $(E_1 \otimes E_2, F)$.

Figure 6.4: A comparison of direct and synchronized product automata of E_1 and E_2.

The direct product $(E_1 \times E_2, F)$ is a universal SDPA, similar to both parent automata. In order to allow both automata in the product to operate independently, the parameter y needs to be renamed to y' in E_2. In the synchronized product $(E_1 \otimes E_2, F)$, however, the parameter y is explicitly not renamed and needs to fulfill the constraints imposed by E_1 and E_2 at the same time. Therefore, $(E_1 \otimes E_2, F)$ only accepts words in which the first letter is largest. $(E_1 \otimes E_2, F)$ is neither universal nor strongly deterministic.

Example 6.7 (intersection in direct and synchronized product)**.** Let there be two PA $A = (M, Q, q_0, \delta_A, F_A)$ and $B = (M, P, p_0, \delta_B, F_B)$ with identical parameter sets $Y_A = Y_B$. Then

$$w \in L((A \times B, F_A \times F_B)) \Leftrightarrow \exists \mu, \mu' \in \Theta : w \in L(A_\mu) \wedge w \in L(B_{\mu'}), \text{ but}$$
$$w \in L((A \otimes B, F_A \times F_B)) \Leftrightarrow \exists \mu \in \Theta : w \in L(A_\mu) \wedge w \in L(B_\mu).$$

In other words, $L((A \times B, F_A \times F_B))$ is the union of all sets $L(A_\mu) \cap L(B_{\mu'})$ where μ and μ' are arbitrary parameter assignments. Meanwhile $L((A \otimes B, F_A \times F_B))$ is the union of all sets $L(A_\mu) \cap L(B_\mu)$, where the same μ is applied in both A and B. As such, $L((A \otimes B, F_A \times F_B)) \subseteq L((A \times B, F_A \times F_B))$.

This concludes the introduction of the synchronized product, one of the building blocks for CFPA. Now we need to define the other component, Skolem automata. Given any complementable PA A, how do we find a PA B such that the synchronized product of A and B is in complementable normal form?

For every PA A and every word $w \in L(A)$, there is a set of "accepting" parameter assignments $T \subseteq \Theta$ such that for every $\mu \in T$, it follows that $w \in L(A_\mu)$. The idea is to find a relation between w and T which can be expressed using another PA B and importantly, which can be generalized to all words $v \in D^*$. The latter condition translates to B being a universal PA. The former condition translates to the following: B should use the same parameters as A. A word $w \in L(A)$ should only be able to complete an accepting run in B when using a parameter assignment from the set T. Not all parameter assignments of T need to imply an accepting run in B, but all parameter assignments from $\Theta \setminus T$ have to lead to non-accepting runs. If B is universal, then for each $w \in L(A)$ there has to be a $\mu \in T$ such that w completes an accepting run in B_μ.

Example 6.8. In the recurring example of A_3 (see figure 6.1), a word $w \in L(A_3)$ is accepted if and only if $\mu(y) \leq min(w)$ and $max(w) \leq \mu(y) + 1$, where $max(w)$ and $min(w)$ denote the largest and smallest letter of w, respectively. A suitable automaton B does not need to maintain the entire range of possible accepting parameter assignments; it is sufficient to isolate a single instance of an accepting parameter assignment for each word. In the example of A_3, a word $w \in L(A_3)$ is always accepted if $\mu(y) = min(w)$, a relation which can be expressed by a PA as depicted in figure 6.2b.

Definition 6.9 (Skolem automaton)**.** Let A be a PA. Let B be a PA such that:

- B is universal,
- the parameters of A are a subset of the parameters of B: $Y_A \subseteq Y_B$, where Y_A and Y_B are the finite sets of parameters used by A resp. B, as defined in notation 3.2,
- for all $w \in L(A)$, if $w \in L(B_\mu)$ for some $\mu \in \Theta$ then $w \in L(A_\mu)$.

Then B is called a Skolem automaton of A.

Finally, we can construct CFPA!

Proposition 6.10. Let $A = (M, Q, q_0, \delta, F_A)$ be a PA that is (without loss of generality) deterministic per assignment. Let $B = (M, P, p_0, \delta', F_B)$ be a Skolem automaton of A. Then the synchronized product $(A \otimes B, F_A \times F_B)$ is equivalent to A and is in complementable normal form. The complement of $L(A)$ is accepted by the set of states $(Q \setminus F_A) \times F_B$.

Proof. Let $w \in L(A)$. Since B is universal, there is a parameter assignment $\mu \in \Theta$ such that $w \in L(B_\mu)$. By the definition of B, this means $w \in L(A_\mu)$, and no complete run of w in A_μ may terminate in a non-accepting state because A is deterministic per assignment. Therefore, $L(A) \subseteq L((A \otimes B, F_A \times F_B))$.

Let now $w \in L((A \otimes B, F_A \times F_B))$. Then there exists a parameter assignment μ such that $w \in L(B_\mu)$ and $w \in L(A_\mu)$. Therefore, $L((A \otimes B, F_A \times F_B)) \subseteq L(A)$. This concludes the proof that $L(A) = L((A \otimes B, F_A \times F_B))$.

In order to prove that $(A \otimes B, F_A \times F_B)$ is in complementable normal form, it is sufficient to prove that $(A \otimes B, (Q \setminus F_A) \times F_B)$ identifies the complement of $L(A)$.

Let $w \in L(A)^c$. As B is universal, there is a parameter assignment $\mu \in \Theta$ such that $w \in L(B_\mu)$. Since A is deterministic per assignment, w completes a run in A_μ which terminates in a non-accepting state since $w \notin L(A)$. Therefore, w completes a run in $(A \otimes B)_\mu$ which terminates in $(Q \setminus F_A) \times F_B$.

Vice versa, let a word w terminate in $(p, b) \in (Q \setminus F_A) \times F_B$ for some parameter assignment μ. Thus, $w \in L(B_\mu)$. If w were in $L(A)$, then the the third condition would force the run of w in A_μ to terminate in an accepting state. However, because w terminates its run in A_μ in a non-accepting state, w cannot lie in $L(A)$. □

An additional restriction has snuck into this proposition that has not been mentioned before: We demand A to be deterministic per assignment.

A construction using the synchronized product with a Skolem automaton will only work if the PA A to be complemented allows runs to terminate in non-accepting states if the chosen parameter assignment does not lead to acceptance. In A_3, for instance, there are no non-accepting states and all non-accepting runs die. This is a problem, because the runs of words $w \in L(A_3)^c$ will die in any synchronized product, as well, and never terminate in a state identifying the word correctly as part of the complement. A PA needs to be at minimum complete per assignment for this method to work.

There is reason to restrict this condition further and demand the PA A to be deterministic per assignment, although strictly speaking, it is not necessary: This way, scenarios in which a word w may complete both an accepting and a non-accepting run in A_μ, due to A_μ being non-deterministic, are avoided. If A is not deterministic per assignment, we need to demand that for each $w \in L(A)$, if $w \in L(B_\mu)$ then *all paths* of w in $L(A_\mu)$ need to terminate in accepting states. Otherwise, words of $L(A)$ will incorrectly be identified as part of the complement. This additional condition is complicated, while determinism per assignment can always be achieved algorithmically, so the latter is preferred.

Example 6.11. We will walk through the construction of a CFPA step-by-step. Consider A_2, the automaton which accepts all words in which the last letter is largest:

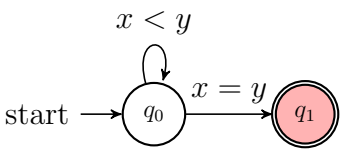


Figure 6.5: A_2.

First, we need to transform A_2 into a PA that is deterministic per assignment. Since A_2 is relatively simple (no two transitions exiting the same state can be viable at the same time), we can do this by simply adding a third state and redirecting all runs that would otherwise die:

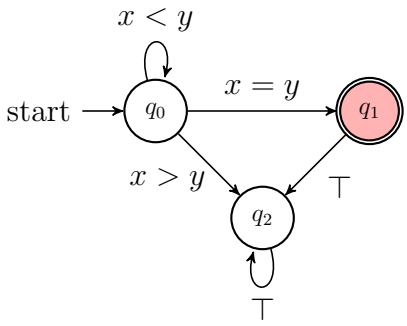
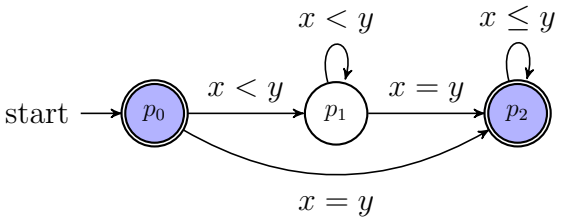


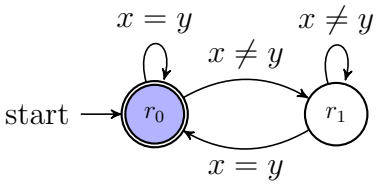
Figure 6.6: $\langle A_2 \rangle$.

We call this new automaton $\langle A_2 \rangle$.

$\langle A_2 \rangle$ has one parameter y. Given a word $w \in L(\langle A_2 \rangle)$, which parameter assignments will allow w to complete an accepting run, and can the relationship between w and those parameter assignments be generalized to all words in D^*? There are two possible avenues, as w will complete an accepting run in $\langle A_2 \rangle_\mu$ if two conditions are met: $\mu(y)$ has to be the last letter in w, and $\mu(y)$ has to be the maximum of all letters in w. On its own, either of these conditions can express the "suitability" of a parameter assignment: $\mu(y)$ is a suitable parameter value for the word $w \in L(\langle A_2 \rangle)$ if it corresponds to the last letter, and $\mu(y)$ is a suitable parameter value for the word $w \in L(\langle A_2 \rangle)$ if it corresponds to the largest letter. Moreover, either of these conditions can be generalized to all words in D^*, as every non-empty word has a largest element and a last element. We will keep developing both approaches in parallel: B_1 accepts a word if $\mu(y)$ corresponds to its largest letter, and B_2 accepts a word if $\mu(y)$ corresponds to its last letter.



(a) B_1.



(b) B_2.

Figure 6.7: Two possible Skolem automata.

Now, we can construct the synchronized product sets $\langle A_2 \rangle \otimes B_1$ and $\langle A_2 \rangle \otimes B_2$. Let $F_A = \{q_1\}$ be the set of accepting states of $\langle A_2 \rangle$, $F_{B_1} = \{p_0, p_2\}$ be the set of accepting states of B_1 and $F_{B_2} = \{r_0\}$ be the set of accepting states of B_2.

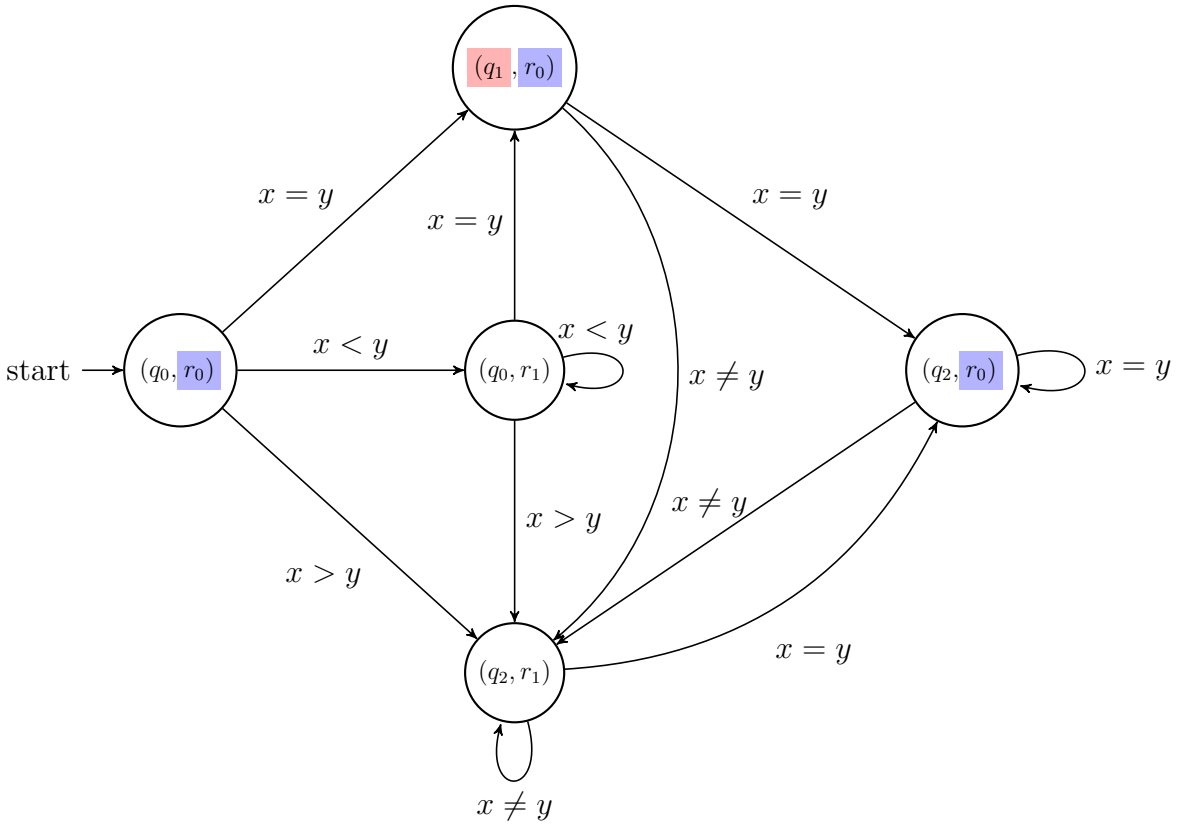


Figure 6.8: $(\langle A_2 \rangle \otimes B_2, F_A \times F_{B_2})$, a CFPA.

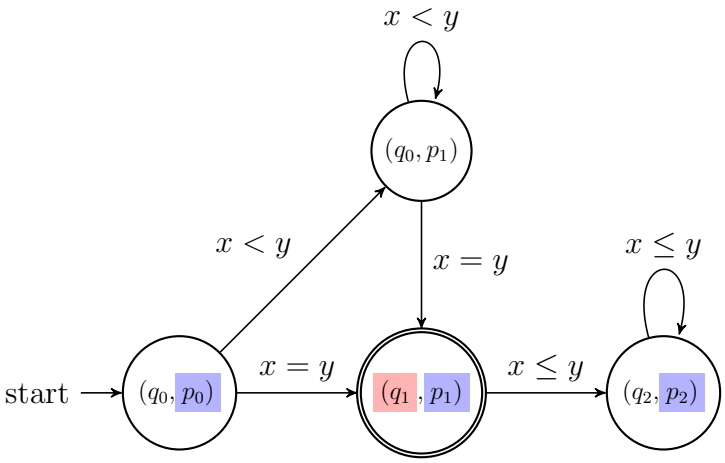


Figure 6.9: $(\langle A_2 \rangle \otimes B_1, F_A \times F_{B_1})$, a CFPA.

In the figures 6.9 and 6.8, the accepting state of $\langle A_2 \rangle$ has been colored red, and the accepting states of B_1 and B_2 have been colored blue. Unreachable states and unsatisfiable transitions have been removed to improve comprehensability. Note that, despite being derived differently, both automata are equivalent.

If the run of a word terminates in a product state that is both red and blue, the word is part of $L(A_2)$. If the run terminates in a product state that is blue but not red, the word is part of $L(A_2)^c$.

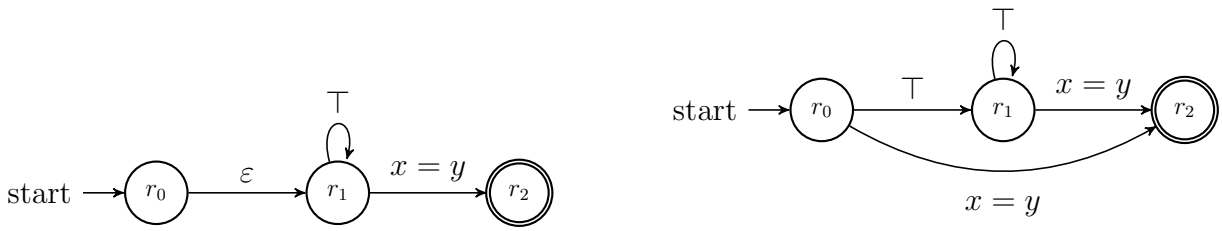
Remark 6.12. Skolem automata do not have to be deterministic per assignment. In fact, they can be neither complete per assignment nor conclusive per assignment. Consider the Skolem automata B_1 and B_2 in figure 6.7. Obviously B_1 is not complete per assignment, because every run dies if $\mu(y)$ is smaller than the current letter. But is there a simpler way to construct B_2, which is unnecessarily deterministic per assignment?

(a) B_2 as seen in example 6.11.

(b) An alternative to B_1? No!

Figure 6.10: Illustrating the importance of universality, which makes Skolem automata more more convoluted as the empty word needs to be respected.

At first glance, the PA in figure 6.10b appears to be a less convoluted alternative to B_2, but it is not a universal PA because the empty word does not complete an accepting run! In order to accept the empty word, we can add a new initial state with an ε-transition and then eliminate the ε-transition, because the synchronized product construction has not been defined for PA with ε-transitions. This can be seen in figure 6.11.



(a) Adding an ε-transition.

(b) The resulting PA B_2' is universal and fulfills the same purpose as B_2.

Figure 6.11: Modifying the PA in figure 6.10b to accommodate the empty word.

The PA B_2' in figure 6.11 is universal, however it has three states in contrast to B_2's two states. But does this necessarily translate to an increase in the number of states in the synchronized product? After all, B_1 also has three states, but its synchronized product with $\langle A_2 \rangle$ has fewer states than the synchronized product of B_2 and $\langle A_2 \rangle$. Maybe there is a benefit to Skolem automata that are not complete per assignment.

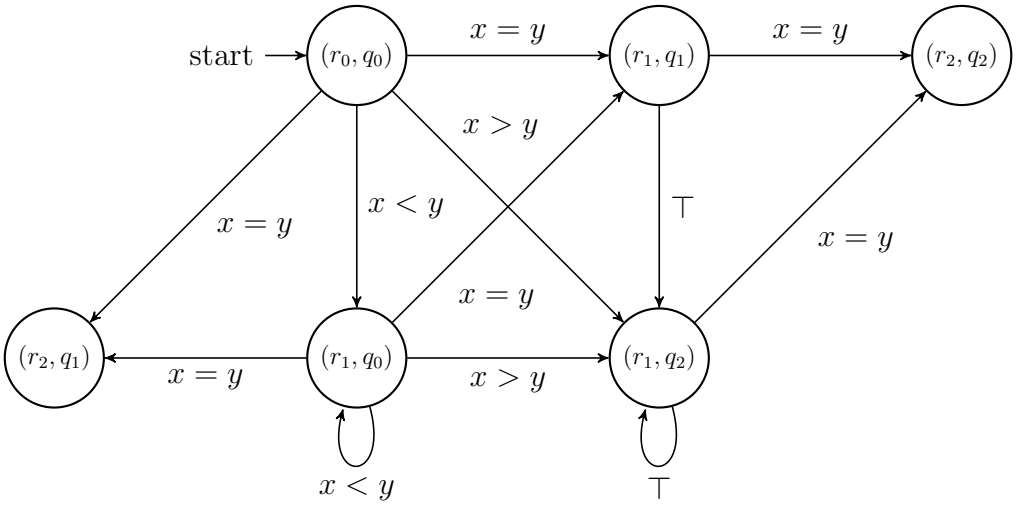


Figure 6.12: The synchronized product of B_2' and $\langle A_2 \rangle$.

As can be seen in figure 6.12, the synchronized product of B_2' and $\langle A_2 \rangle$ has six states, which is worse than the five states of $(\langle A_2 \rangle \otimes B_2, F_A \times F_{B_2})$. We can conclude that there is no inherent benefit to Skolem automata which are incomplete per assignment. Nor is there a benefit to Skolem automata that are deterministic per assignment, as $(\langle A_2 \rangle \otimes B_1, F_A \times F_{B_1})$ outperforms $(\langle A_2 \rangle \otimes B_2, F_A \times F_{B_2})$ with regard to the number of states.

After concluding the description of our method, this section closes with a brief investigation into the applicability of the same. According to theorem 6.3, for every complementable PA, there is an equivalent CFPA. A similarly structured proof can show the existence of a Skolem automaton for every complementable PA:

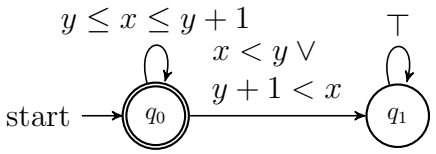
Proposition 6.13. For every complementable PA A that is deterministic per assignment, a suitable Skolem automaton B exists.

Proof. Let $A = (M, Q, q_0, \delta_A, F_A)$ and $A^c = (M, P, p_0, \delta_C, F_C)$ be a complementable PA and its complement, respectively. Without loss of generality, assume that the finite sets of parameters used by A and A^c are disjoint, $Y_A \cap Y_{A^c} = \varnothing$. All parameters used by A and A^c are contained in the finite union $Y' = Y_A \cup Y_{A^c}$.

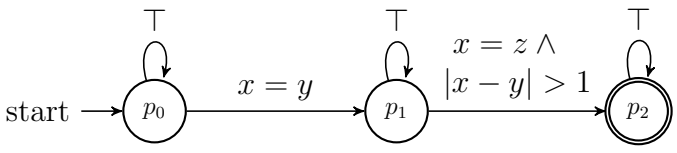
Now form the union of A and A^c using ε-transitions and a new state r that is neither a state of A nor a state of A^c. The resulting PA $B = (M, Q \cup P \cup \{r\}, r, \delta_A \cup \delta_C \cup \{(r, \varepsilon, q_0), (r, \varepsilon, p_0)\}, F_A \cup F_C)$ is universal. Let $w \in L(A)$ and μ be a parameter assignment such that a run of w in B terminates in an accepting state $q \in F_A \cup F_C$. Then q cannot lie in F_C, because $w \in L(A)$. Since the run of w in B_μ has to terminate in a state of F_A, $w \in L(A_\mu)$.

Finally, since $Y_A \subseteq Y'$, we can set $Y_A := Y'$ (Y' is still a finite set, and we can assume that any parameters $y \in Y' \setminus Y_{A^c}$ simply do not occur). Therefore, B fulfills all conditions defined in definition 6.9. □

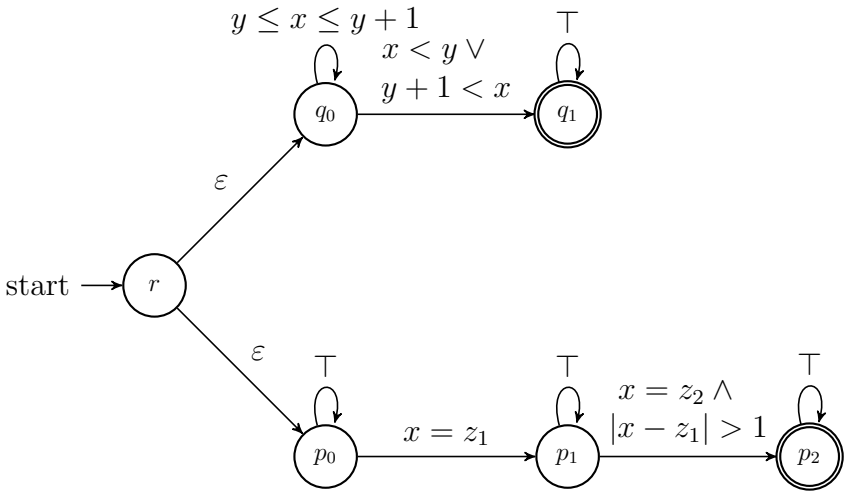
Applicability of the method hinges solely on the existence of a Skolem automaton, meaning that in combination with the algorithm achieving determinism per assignment, the method can be applied to every complementable PA to obtain a CFPA.



(a) A PA for L_3 that is deterministic per assignment.



(b) A PA for $L(A_3)^c$.



(c) Union construction, obtaining a Skolem automaton for A_3.

Figure 6.13: The construction of proposition 6.13, again demonstrated using L_3.

Example 6.14. Figure 6.13 illustrates the mechanism of proposition 6.13: We combine a PA for L_3 with an arbitrary PA identifying L_3^c.

The previous proposition proves the existence of a Skolem automaton for any A which is deterministic per assignment, but does not provide a method for constructing the Skolem automaton. After all, the automaton in the proof is obtained using the complement automaton of A, and if the complement is already known, construction of a CFPA is obsolete.

So, given a pair of PA A and B, can we verify whether B is a Skolem automaton of A? A Skolem automaton has to fulfill three properties, which each can be confirmed with widely differing degrees of difficulty: The universality problem is undecidable, while $Y_A \subseteq Y_B$ is easily checked. The third property can be verified as well, hinging on the complexity of the non-emptiness problem.

Proposition 6.15. Given a PA $A = (M, Q, q_0, \delta_A, F_A)$ that is deterministic per assignment and a universal PA $B = (M, P, p_0, \delta_B, F_B)$, it can be decided whether B is a Skolem automaton of A.

Proof. The first two properties of Skolem automata are easy to verify, since we already assume that B is universal and $Y_A \subseteq Y_B$ can be checked quickly. We only need to

prove that for every $w \in L(A)$ and every parameter assignment μ, $w \in L(B_\mu)$ implies $w \in L(A_\mu)$.

The condition is breached if there is a $w \in D^*$ and a μ such that $w \in L(A) \cap L(B_\mu) \cap L(A_\mu)^c$. We can construct a PA for $L(A) \cap \bigcup_{\mu\in\Theta}(L(B_\mu) \cap L(A_\mu)^c)$ using both a direct and a synchronized product, and then verify whether the language is empty.

The language $\bigcup_{\mu\in\Theta}(L(B_\mu) \cap L(A_\mu)^c)$ corresponds to the synchronized product $(A \otimes B, (Q \setminus F_A) \times F_B)$, as seen in example 6.7. We can then express the intersection using a direct product with A: $L(A)\cap\bigcup_{\mu\in\Theta}(L(B_\mu)\cap L(A_\mu)^c) = (A\times(A\otimes B, (Q\setminus F_A)\times F_B), F_A \times (Q \setminus F_A) \times F_B)$. If $(A \times (A \otimes B, (Q \setminus F_A) \times F_B), F_A \times (Q \setminus F_A) \times F_B)$ is empty, then B is a Skolem automaton of A. □

In definition 6.9 and propositions 6.10, which define and describe the construction of CFPA using Skolem automata, the PA A is implied but not required to be complementable. This opens up an interesting new way of confirming whether a PA is complementable: A PA (assumed to be deterministic per assignment) is complementable if and only if a Skolem automaton exists.

6.3 Closure and Decision Properties

Similar to other classes of PA, we will now quickly investigate the properties of CFPA regarding decision problems and Boolean operations.

First, it is refreshing to see the set of complement-identifying states F_C does not need to be known along with a CFPA. As long a PA is known to be in complementable normal form, this information can be reconstructed.

Theorem 6.16. Given a CFPA $A = (M, Q, q_0, \delta, F)$, it is possible to identify a set of states $F_C \subseteq Q$ such that $A^c = (M, Q, q_0, \delta, F_C)$ identifies the complement of A.

Proof. It is sufficient to identify the set of all states that cannot be reached by words in $L(A)$. As seen in proposition 4.17, we can use a direct product construction to observe whether two states of a PA can be reached by the same word.

Let $A \times A$ be the set of direct product automata of A with itself, using disjoint parameter sets. Then, a state q lies in F_C if and only if the PA $(A \times A, F \times \{q\})$ is empty. □

This method identifies the largest subset $F_C \subseteq Q$ such that $A^c = (M, Q, q_0, \delta, F_C)$ identifies the complement of A, but F_C does not need to be the only set of states with this property.

Theorem 6.16 significantly simplifies the complementation of CFPA, since the set of complement-accepting states F_C does not need to be communicated along with the CFPA. Therefore, the decision properties of CFPA are largely similar to those of SDPA.

Theorem 6.17. The universality, containment and equivalence problems are decidable for CFPA.

Proof. The proof for the containment and equivalence problems is analogous to the proof for SDPA, since the latter only relied on the fact that SDPA can be easily complemented. Since a PA is universal if and only if its complement is empty, the universality problem is also decidable. □

CFPA must be closed under all Boolean operations, because complementable PA are closed under all Boolean operations and every complementable PA is equivalent to some CFPA. Therefore, the following theorem is not concerned with existence but with computability: The proof contains a description of how to derive CFPA that are equivalent to the complement, union and intersection of two given CFPA.

Theorem 6.18 (Boolean operations on CFPA)**.** Given two CFPA $A_1 = (M, Q, q_0, \delta_1, F_1)$ and $A_2 = (M, P, p_0, \delta_2, F_2)$, CFPA that represent the union $L(A_1) \cup L(A_2)$, the intersection $L(A_1) \cap L(A_2)$ and the complement $L(A_1)^c$ can be computed algorithmically.

Proof. Let $C_1 \subseteq Q$ be the set of complement-accepting states of A_1, i. e. $L(A_1)^c = L((M, Q, q_0, \delta_1, C_1))$, and let $C_2 \subseteq P$ be the set of complement-accepting states of A_2. Both C_1 and C_2 can be identified according to theorem 6.16, and $(M, Q, q_0, \delta_1, C_1)$ is obviously a CFPA. This concludes the statement for the complementation of CFPA.

CFPA are by definition complete, as every word will complete a run in either an accepting or a complement-accepting state. Therefore, the direct product $A_1 \times A_2$ can be employed to construct CFPA for $L(A_1) \cup L(A_2)$ and $L(A_1) \cap L(A_2)$.

The language $L(A_1) \cup L(A_2)$ is represented by $(A_1 \times A_2, F_1 \times P \cup Q \times F_2)$. The PA is a CFPA because $(A_1 \times A_2, C_1 \times C_2)$ identifies the complement $(L(A_1) \cup L(A_2))^c$.

The language $L(A_1) \cap L(A_2)$ is represented by $(A_1 \times A_2, F_1 \times F_2)$. The PA is a CFPA because $(A_1 \times A_2, C_1 \times P \cup Q \times C_2)$ identifies the complement $(L(A_1) \cap L(A_2))^c$.

□

Chapter 7

What Have We Learned?

7.1 Revisiting Dijkstra

With a whole toolbox of new methods, we can now revisit the motivating example from section 1.1. As a reminder, the PA to be complemented looks like this:

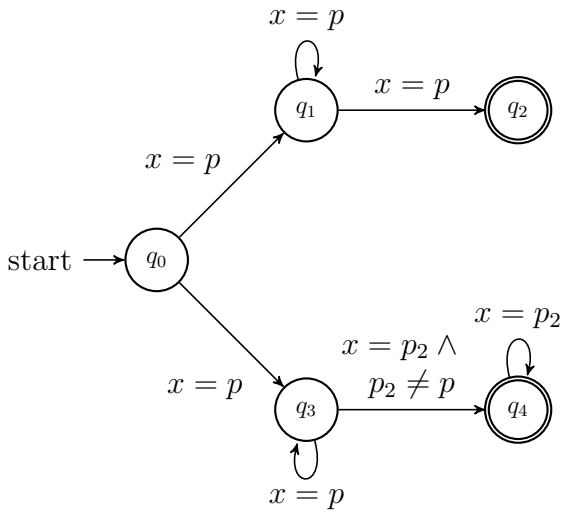


Figure 7.1: A non-deterministic parametrized automaton P that witnesses whether the input ring of processes has exactly one privileged process.

Our goal is to find a CFPA equivalent to P.

In P, the possible system states are represented by strings of variables. A system state has exactly one privileged process if either all variables are identical, or if there is an index i such that the first i variables are assigned a value p, while the remainder of variables are assigned a distinct value p_2. This distinction of accepting scenarios is reflected by P's structure, as P's upper path leading to state q_2 recognizes the former scenario, while the lower path leading to state q_4 identifies the latter. In its current form, P is very comprehensible for human readers.

Unfortunately, P is not deterministic per assignment, which is a prerequisite for applying the method described in section 6.2. This needs to be remedied. For this purpose, the non-determinism upon leaving the initial state q_0 needs to be resolved and a state needs to be added to ensure the automaton is complete per assignment.

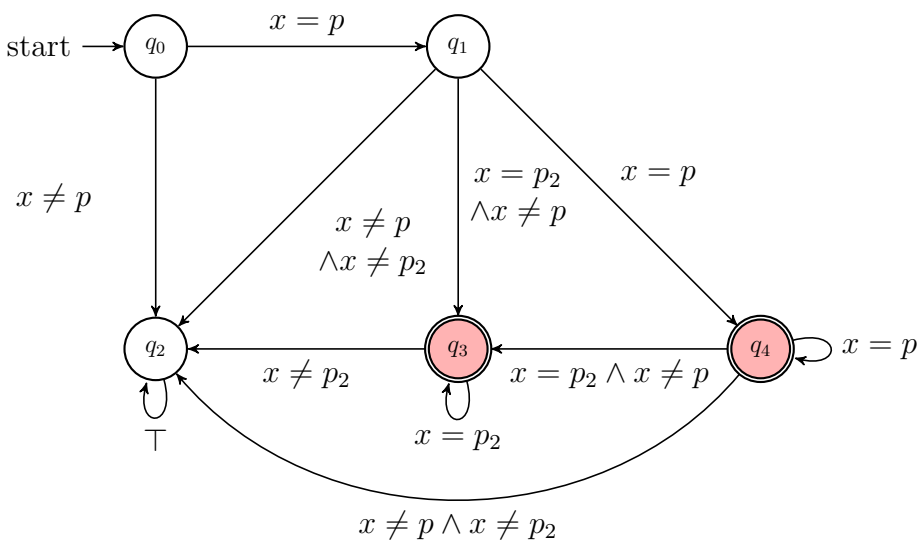


Figure 7.2: $\langle P \rangle$, which is equivalent to P and deterministic per assignment.

We call the new automaton $\langle P \rangle$.

Next, we need to find a suitable Skolem automaton. Clearly, a word accepted by $\langle P \rangle$ is accepted whenever p is assigned the value of the first letter. Therefore, a Skolem automaton B should enforce $\mu(p) = w_1$ for every $w = w_1 w_2 \dots w_n \in L(B_\mu)$. However, this alone is not sufficient to enforce the third condition of definition 6.9. The second parameter p_2 also matters, because the runs of words with a single, privileged process $i \neq 1$ only reach the accepting state q_3 if $\mu(p_2) = w_i$. If a different value for $\mu(p_2)$ is chosen, then the run terminates in q_2, which is a non-accepting state. Since all letters starting from w_i have to be identical, we can set $\mu(p_2)$ to correspond to the last letter, since every non-empty word has a last letter.

Therefore, a Skolem automaton B shall accept words whenever p is assigned the first letter, and p_2 is assigned the last letter.

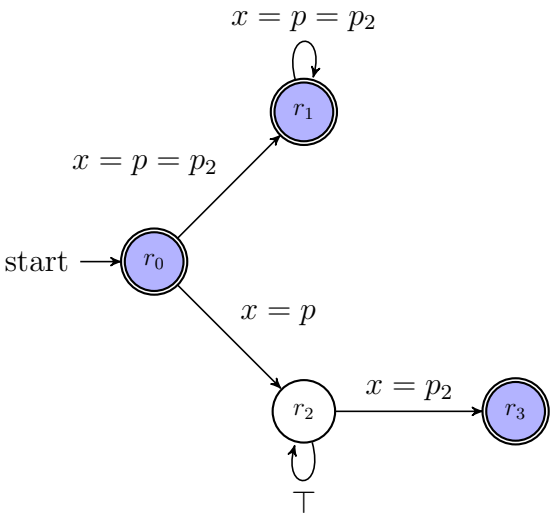


Figure 7.3: A Skolem automaton B for $\langle P \rangle$.

The empty word is accepted with every configuration of parameters. If the word only

consists of one letter, or if all letters in the word are identical, $\mu(p_2)$ will default to $\mu(p)$.

If we then construct the synchronized product of $P' = \langle P \rangle \otimes B$, we will obtain a CFPA for P.

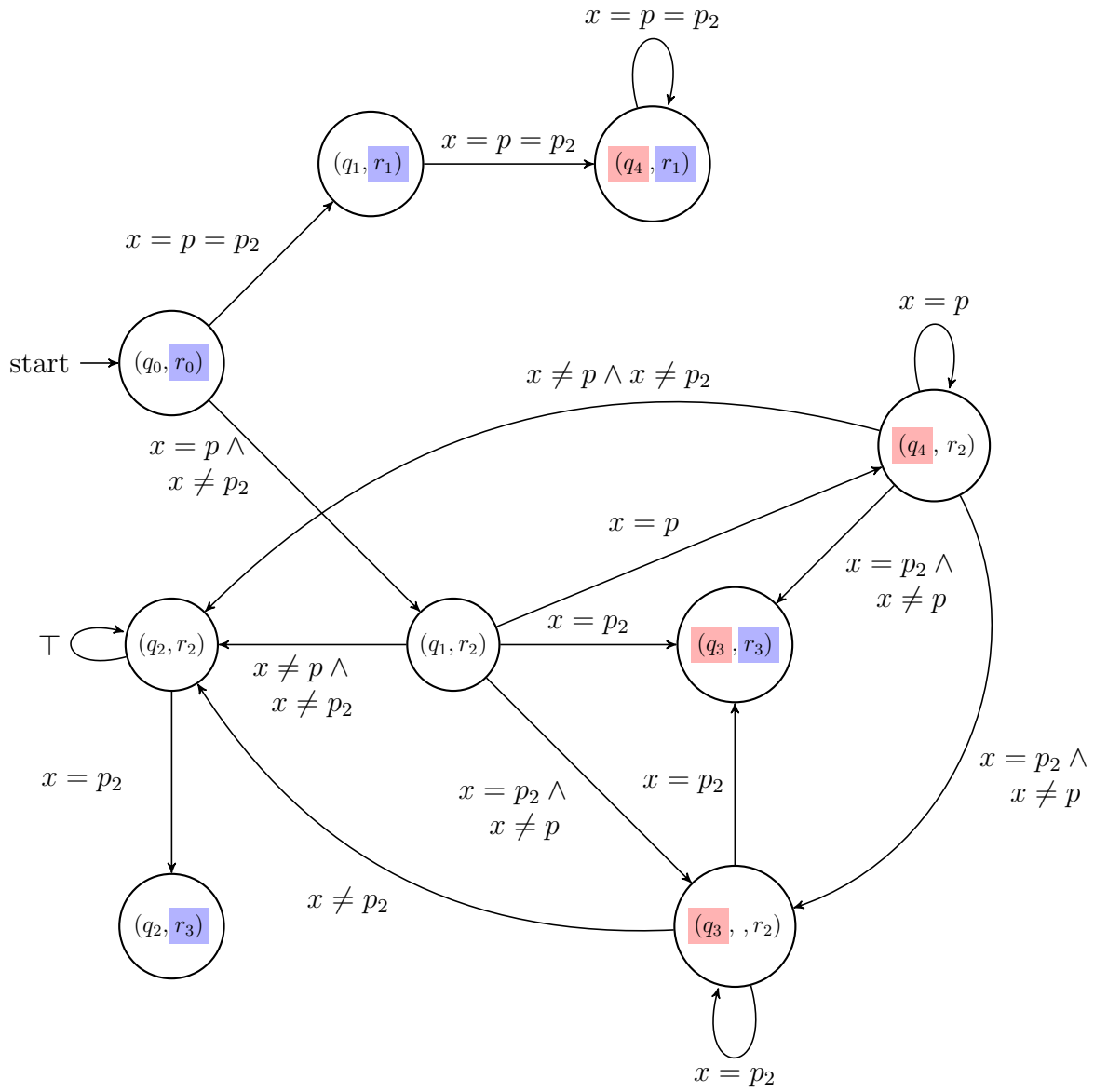


Figure 7.4: A PA (P', F) equivalent to P in complementable normal form, where $F = \{(q_4, r_1), (q_3, r_3)\}$.

In many regards, the resulting synchronized product P' reaches or even overshoots the goals set in section 1.1:

- P' can correctly identify $L(P)^c$ when selecting $F_c = \{(q_0, r_0), (q_1, r_1), q_2, r_3)\}$ as the set of accepting states. It is also capable of identifying the original
- While appearing large and convoluted, P' is smaller than expected. The transformation into $\langle P \rangle$ did not lead to an exponential increase of the number of states, and the synchronized product P' only consists of 9 states instead of the expected

maximum size of 20. The number of states also stays below the limit provided theorem 6.3, which is 11 based on P and the PA P^c manually constructed in figure 1.3. Note that P^c itself is not a CFPA, since the runs of words in which the first letter is privileged always terminate in state q_1, which is a weak state. language when choosing $F = \{(q_4, r_1), (q_3, r_3)\}$.

- P' has been obtained using a method that can be generalized to other complementable PA.
- The correctness of P' can be argued both based on the method, which has been proven to work correctly, and by applying the construction in the proof of theorem 6.16.

7.2 Future research

Over the course of this thesis, we have bumped into a number of open questions that leave room for future research. First and foremost, since the universality problem is undecidable for PA, there cannot be an algorithm that both identifies and then complements complementable PA. It remains an open question whether part of the problem is decidable.

Question 1. Is there a method to identify complementable PA? Or is there a method which can complement PA that are known to be complementable?

Of similar importance is the question whether completeness of PA is decidable. A PA is complete if every word completes a run.

Question 2. Is there an algorithm that correctly identifies complete parametrized automata?

This question is crucial because it plays into two important concepts we have introduced to overcome the limitations imposed by the undecidability of the complementation problem: Strong determinism, and complementable normal form.

Completeness may be easier to prove for conclusive PA, because it is easier to localize instances where a word does not complete a run. In general, a word whose run is aborted using one parameter assignment may still complete a different run using a different parameter assignment. In conclusive PA, we do not have to "backtrack" as much: If a prefix of a word completes a run, a complete run of the word has to include the prefix's run. This leads to the second question, which is a weakened version of the first.

Question 3. Is there an algorithm to correctly identify SDPA?

Such a method exists for variable automata, which allow for tighter control of the parameters which permit a word to complete a run. An equivalent method for PA has not been found yet.

We can pose the same question for PA in complementable normal form.

Question 4. Is there an algorithm that correctly identifies CFPA?

There is also an open gap in the construction of CFPA, as we have not provided a method of finding suitable Skolem automata.

Question 5. Is there a method for obtaining Skolem automata?

A first step might be to identify subclasses of parametrized automata for which the question is easier to answer.

Question 6. Are there reasonable subclasses of PA, SDPA or CFPA for which the completeness problem is decidable?

Trivial answers are deterministic finite-state or symbolic automata.

This thesis barely scratches the surface when it comes to studying the properties of the parameter assignment. We have defined strictness and strictness-when-accepting, but only completed a partial investigation on the prevalence of those concepts.

At first, the existence of PA that cannot be made strict-when-accepting needs to be proven.

Question 7. Is there a language that can be accepted by a PA, but not by a PA that is strict-when-accepting?

A strong candidate is L_1, the language of all unsorted words.

Ultimately, we hope for a connection between complementability and strictness. We posit that both properties coincide.

Question 8. Is the class of PA that can be complemented exactly the class of PA that can be made strict-when-accepting?

A first step might be to investigate the parameter assignment's properties for variable automata.

Question 9. Is there a VA that is not equivalent to a VA that is strict-when-accepting? Is a VA complementable if and only if it can be made strict-when-accepting?

Finally, we note that strictness (as well as strictness-when-accepting) is a syntactic property that is usually more powerful than necessary and does not allow for much nuance. Oftentimes, making a PA strict adds unneeded complexity while making the PA less comprehensible and harder to work with. There are two possible angles from which we can close in on a more fine-grained characterization. For one, the strictness criterion could be constrained to only the parameters that actually occur during a run. A lot of bloat in strict automata is caused because parameters that are not relevant to a word's run need to be taken care of. Another stems from the fact that transitions often permit ranges of parameters, instead of assigning a parameter directly, and these ranges may get more refined as a word's run progresses.

Therefore, another challenge for future researchers lies in finding meaningful, more fine-grained descriptions of the properties of the parameter assignment, and how these properties can be exploited when complementing parametrized automata.

This work does not touch on the subject of parametrized transducers [17] or parametrized automata on infinite words [2], either of which warrant a separate investigation.

7.3 Conclusions

In this thesis, we have laid the foundation for a theory of parametrized automata and obtained a solid grasp for the challenges regarding the complementation of PA. We have shown PA are not closed under complementation, and an algorithm that either complements a given PA, or returns that the PA is not complementable, does not exist.

We have identified SDPA and CFPA as subclasses of PA that can be easily complemented. Most of the important decision problems are decidable for SDPA and CFPA, and either are closed under all Boolean operations. While SDPA have shown themselves to be elusive, as they do not encompass all complementable PA, we have shown that every complementable PA can be transformed into a CFPA, and proposed a method by which such a transformation can be achieved.

7.4 Acknowledgments

I would like to express my deepest gratitude to Prof. Dr. Philipp Rümmer for his exceptional mentorship, expertise and guidance. His insights and feedback have been instrumental in shaping this thesis, and I especially enjoyed our regular exchange and discussions of new ideas.

This thesis could not have been possible without the efforts of Prof. Dr. Clara Löh. I'm extremely grateful for her willingness to battle bureaucracy on my behalf, and her competence and guidance both within and outside of the scope of this thesis.

I would also like to extend my sincere thanks to my colleagues Dr. Julie Cailler and Roland Herrmann for their valuable advice and support throughout this project.

Lastly, I am grateful to my friends and family. I would like to especially thank Alexander Alber for his kind, patient, unwavering presence in my life, my parents for being the best parents ever and my cats for not deleting this entire thesis by accidentally stepping on the wrong keys.

7.5 Declaration as per § 20 (5) of the Study Regulations

I hereby declare that this master thesis is my own work, without the use of any sources or aids other than those stated. This work has not been submitted for an academic degree at any other institution. I verify that the printed copies and electronic version of this thesis are identical, that I have been educated on proper academic conduct and referencing, and that I am aware of the legal consequences outlined in § 26 (5) of the study regulations.

Ich erkläre hiermit, dass ich diese Masterarbeit eigenständig verfasst habe und nur die angegebenen Quellen und Hilfsmittel verwendet wurden. Diese Arbeit wurde nicht bereits an einer anderen Hochschule zur Erlangung eines akademischen Grades eingereicht. Die vorgelegten Druckexemplare sind identisch zur vorgelegten elektronischen Version der Arbeit. Ich wurde über wissenschaftlich korrektes Arbeiten und Zitieren aufgeklärt und habe Kenntnis von den in § 26 Abs. 5 der Studienordnung vorgesehenen Rechtsfolgen.

Glossary

ε-transition A transition that does not consume a letter. 17

alphabet A non-empty set. The elements of this set are called letters. 10

CFPA A parametrized automaton in complementable normal form. 67

complement The complement of a language A is the language of all words that are not part of A. 11

complement automaton In the context of automata, an automaton C is the complement automaton of an automaton A if C accepts exactly the words that A does not accept. 18

complementable A parametrized automaton A is complementable if there exists another parametrized automaton C fulfilling $L(C) = L(A^c)$. 39

complementable normal form A parametrized automaton is in complementable normal form if it can be transformed into its own complement automaton by selecting a different set of accepting states. 67

complete A parametrized automaton is complete if every word completes at least one run. 47

concatenation The operation of appending one word to another to create a new word. 11

conclusive A parametrized automaton is conclusive if every word completes at most one run. 47

containment problem A decision problem preoccupied with whether a language is a subset of another language. 20

De Morgan's laws For sets A and B, it holds that $(A \cup B)^c = A^c \cap B^c$ and $(A \cap B)^c = A^c \cup B^c$. 39

death of a run The abortion of a run in case a state without viable exit transitions is reached without having terminated. 18

decidable A decision problem (represented as a set of yes-cases) is decidable if there exists an algorithm that terminates on every input, and correctly decides whether the input belongs to the set or not. 19

decision problem A yes-or-no question. 19

deterministic A finite-state, register, variable or symbolic automaton is deterministic if every word completes exactly one run in the automaton. 18

deterministic per assignment A parametrized automaton is deterministic per assignment if for every word and every parameter assignment, there is exactly one complete run. 34

determinizable A parametrized automaton is determinizable if it is equivalent to an SDPA. 47

direct product For two parametrized automata A and B, the direct product set $A \times B$ is a set of automata that can simulate two independent runs of a word in A and B simultaneously. 36

empty word A word that does not contain letters, commonly denoted ε. 10

equivalence problem A decision problem preoccupied with whether two languages are equivalent. 20

equivalent states Two states q, p of a symbolic automaton are equivalent if they fulfill the same "purpose" in the automaton, meaning that if the run of a word starting in q terminates in an accepting state, a run of the word starting in p also has to terminate in an accepting state, and vice versa. In parametrized automata, two states are equivalent if they are equivalent for every fixed parameter assignment. 89

evaluation function A function that maps formulae to the Boolean values *true* and *false* according to the semantics defined by a structure and syntax defined by a signature. 15

finite-state automaton An abstract machine that identifies a regular language. 17

formal language A set of words. 11

formula A logical statement adhering to strict syntactic rules. 14

intersection The intersection of two languages A and B is the language that consists of all words that simultaneously occur in both A and in B. 11

isomorphism of parametrized automata Two parametrized automata are isomorphic if for every parameter assignment, the resulting symbolic automata are isomorphic. 91

isomorphism of symbolic automata Two symbolic automata are isomorphic if they accept the same language and one can be transformed into the other by relabeling the states, up to equivalence of the guards. 90

membership problem A decision problem preoccupied with whether a word is contained in a language. 20

minimum of a parametrized automaton The minimum of a parametrized automaton A is obtained by eliminating redundant, equivalent states. 90

minimum of a symbolic automaton The minimum of a symbolic automaton A is obtained by eliminating redundant, equivalent states. It is the automaton with the least number of states that is equivalent to A. 90

non-emptiness problem A decision problem preoccupied with whether a language is not empty. 19

NP The complexity class containing all decision problems solvable by a non-deterministic Turing machine in polynomial time. 20

P The complexity class containing all decision problems solvable by a deterministic Turing machine in polynomial time. 20

parameter assignment A function that maps the parameters of a parametrized automaton to concrete values. 28

parametrized automaton An automaton suited for languages over infinite alphabets, whose transitions are labeled with logical formulae that also contain a finite number of non-reassignable parameters. 27

power set The power set of a set S is the set of all subsets of S, including S itself and the empty set. 18

PSPACE The complexity class containing all decision problems solvable by a deterministic Turing machine in polynomial space. 20

reachability problem A decision problem preoccupied with whether for a state of an automaton, there exists a word such that a run of that word enters that state. 20

register automaton An automaton with finite, reassignable storage space for input letters which can be compared to upcoming letters for equality. Suitable for languages over infinite alphabets. 21

regular expression An expression that constructs a language. Regular expressions are defined inductively from smaller regular expressions, where base cases are individual letters and the empty language and permitted operations are concatenation, union and Kleene closure. 12

regular language A formal language that can be defined by a regular expression. 12

run The sequence of states and transitions traversed by a word in an automaton. If picturing automata as directed graphs, a run is a path starting in the initial state. 17

SDPA A strongly deterministic parametrized automaton. 47

semidecidable A decision problem (represented as a set of yes-cases) is semidecidable if there exists an algorithm that terminates for every input that is part of the set. 19

set of parameters Each parametrized automaton makes use of a finite number of non-reassignable parameters that can occur in the transition guards. 27

signature A tuple defining the symbols (and their arity) that may occur in a logical formula. 13

Skolem automaton A Skolem automaton for a fixed PA A is a universal parametrized automaton which accepts a word in $L(A)$ only if the chosen parameter assignment leads exclusively to accepting runs in A. 70

strict A parametrized automaton is strict if for every non-empty word there is at most one parameter assignment such that the word completes a run. 58

strict-when-accepting A parametrized automaton is strict-when-accepting if for every non-empty accepted word there is exactly one parameter assignment such that the word completes an accepting run. 58

strongly deterministic A parametrized automaton is strongly deterministic if every word completes exactly one run. 47

structure A tuple defining how to interpret a formula. 14

subformula A formula that occurs in the construction of a larger formula. 14

symbolic finite automaton An automata class suitable for infinite alphabets where transitions are labeled with logical formulae instead of letters. 25

synchronized product For two parametrized automata A and B, the synchronized product $A \times B$ is a set of automata that can simulate two runs of a word in A and B simultaneously. The parameter sets of A and B are permitted to intersect, therefore the runs are not independent. 69

theory Can formalize complex structures to allow for logical reasoning. 16

union The union of two languages A and B is the language that contains exactly all words of A and all words of B. 11

universal language The language containing all words (over a fixed alphabet D). 11

universality problem A decision problem preoccupied with deciding whether a language is equivalent to the universal language. 19

variable assignment A function that maps the variables occurring in formulae to concrete values. 14

variable finite automaton An automaton suitable for languages over infinite alphabets with finite, non-reassignable storage space for input letters which can be compared for equality and inequality to upcoming letters. 22

viable A transition is viable if it permits the current letter to exit the current state. 18

word A finite sequence of letters. 10

Bibliography

[1] Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007.

[2] J. R. Büchi. *On a Decision Method in Restricted Second Order Arithmetic*. Stanford: Stanford University Press, 1962.

[3] Alonzo Church. Edward f. moore. gedanken-experiments on sequential machines. automata studies, edited by c. e. shannon and j. mccarthy, annals of mathematics studies no. 34, litho-printed, princeton university press, princeton1956, pp. 129–153. *Journal of Symbolic Logic*, 23(1):60–60, 1958.

[4] Loris D'Antoni, Tiago Ferreira, Matteo Sammartino, and Alexandra Silva. Symbolic register automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2019.

[5] Loris D'Antoni and Margus Veanes. Minimization of symbolic automata. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 541–554. ACM, 2014.

[6] Loris D'Antoni and Margus Veanes. Extended symbolic finite automata and transducers. *Formal Methods Syst. Des.*, 47(1):93–119, 2015.

[7] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 47–67. Springer, 2017.

[8] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[9] Diego Figueira and Anthony Widjaja Lin. Reasoning on data words over numeric domains. In Christel Baier and Dana Fisman, editors, *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, pages 37:1–37:13. ACM, 2022.

[10] Orna Grumberg, Orna Kupferman, and Sarai Sheinvald. Variable automata over infinite alphabets. In Adrian-Horia Dediu, Henning Fernau, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications, 4th International Conference, LATA 2010, Trier, Germany, May 24-28, 2010. Proceedings*, volume 6031 of *Lecture Notes in Computer Science*, pages 561–572. Springer, 2010.

[11] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition.* Pearson international edition. Addison-Wesley, 2007.

[12] Emil Indzhev and Stefan Kiefer. On complementing unambiguous automata and graphs with many cliques and cocliques. *Inf. Process. Lett.*, 177:106270, 2022.

[13] Artur Jez, Anthony W. Lin, Oliver Markgraf, and Philipp Rümmer. Decision procedures for sequence theories. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 18–40. Springer, 2023.

[14] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.

[15] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[16] Albert R. Meyer and Larry J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *13th Annual Symposium on Switching and Automata Theory, College Park, Maryland, USA, October 25-27, 1972*, pages 125–129. IEEE Computer Society, 1972.

[17] Mehryar Mohri. Finite-state transducers in language and speech processing. *Comput. Linguistics*, 23(2):269–311, 1997.

[18] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004.

[19] Ed. Q. H. Taub. Design of computers—theory of automata and numeral analysis,. *John von Neumann – Collected Works*, 5:288–328, 1963.

[20] Hiroshi Sakamoto and Daisuke Ikeda. Intractability of decision problems for finite-memory automata. *Theor. Comput. Sci.*, 231(2):297–308, 2000.

[21] Michael Sipser. *Introduction to the theory of computation.* PWS Publishing Company, 1997.

[22] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*, pages 498–507. IEEE Computer Society, 2010.

Appendix A

Minimization

There is a problem that has, so far, not been adressed: Many of the operations performed on PA (such as both kinds of product constructions) lead to an increase in the number of states. Sometimes, this increase is unavoidable. For example, a CFPA will often need to have more states than an equivalent PA, since the latter does not need to concern itself with identification of the complement. Another example is determinization per assignment, as described in proposition 3.9. The algorithm can lead to a worst case exponential blowup of the number of states, yet determinism per assignment is a necessary requirement for the construction of CFPA.

Nonetheless, there is an increasing need for a method to simplify PA while preserving important properties. Working with large automata both increases computation time and decreases comprehensibility. Consider figures 6.8 and 6.9: Although the PA look very different, both are in complementable normal form and describe the same language. Ideally, there would be a simplification method that yields a smaller, "standardized" version of either PA.

On the search for such a method, we turn to minimization, which is another concept that can be lifted from finite and symbolic automata. The following results will lean heavily on [5].

In symbolic automata, minimization has very pleasant properties: For every SFA A, there exists a SFA $Min(A)$ which is equivalent to A, has a minimal number of states and can be computed algorithmically. If two SFA A and B are equivalent, i. e., describe the same language, then $Min(A)$ and $Min(B)$ will be identical up to relabeling of states and equivalence of transitions. In this situation, we call $Min(A)$ and $Min(B)$ isomorphic. Isomorphism is "stronger" than equivalence because the latter does not care about the structure of the automata.

Transferring minimization and all underlying and accompanying concepts to PA requires thorough preparation. We will only give a brief introduction to minimization of SFA, omitting most proofs, and then investigate how the concept may be applied to PA.

Definition A.1 (equivalence of states in SFA)**.** Let $A = (M, Q, q_0, \delta, F)$ be a SFA and q be a state of A. Let $L^q(A)$ describe the set of words that terminate in an accepting state of A when starting in q, meaning that a word $w = (w_1, \ldots, w_n)$ is part of $L^q(A)$ if there is a sequence of states and transitions $(q, \phi_1, q_1), \ldots, (q_{n-1}, \phi_n, q_n)$ such that $q_n \in F$ is an accepting state and ϕ_i evaluates to true when plugging in w_i for all $i = 1, \ldots, n$. Two states $q, p \in Q$ are equivalent, $q \sim p$, if $L^q(A) = L^p(A)$.

For example, $L^{q_0}(A) = L(A)$ describes the entire language of an SFA. The relation $\sim$ is an equivalence relation. An SFA can be minimized by eliminating redundant, equivalent states.

Theorem A.2 (minimization of SFA)**.** For any deterministic, normalized SFA $A = (M, Q, q_0, \delta, F)$, let $Min(A) = (M, Q_{/\sim}, q_{0/\sim}, \delta_{/\sim}, F_{/\sim})$ be the minimum of A, where

- $Q_{/\sim}$ is the quotient set of Q by $\sim$,
- $q_{0/\sim}$ is the equivalence class of q_0,
- $F_{/\sim}$ is the quotient set of F by $\sim$, i. e., the set of equivalence classes of states in F,
- and $\delta_{/\sim} = \{(q_{/\sim}, \bigvee_{\varphi \in S} \varphi, p_{/\sim}) \mid q_{/\sim}, p_{/\sim} \in Q_{/\sim}, S = \{\varphi \mid (q, \varphi, p) \in Q, q \in q_{/\sim}, p \in p_{/\sim}\}\}$.

It holds that $L(A) = L(Min(A))$, and $Min(A)$ is deterministic and normalized.

The minimum of an SFA is unique up to relabeling of states and equivalence of transitions: If there are two SFA A, B where $L(A) = L(B)$, then $Min(A)$ and $Min(B)$ are isomorphic.

Proof. See [5, section 2, theorem 2]. □

Definition A.3 (isomorphism of SFA)**.** Two SFA $A = (M, Q, q_0, \delta_A, F_A)$, $B = (M, P, p_0, \delta_B, F_B)$ are isomorphic if $L(A) = L(B)$ and there is a bijection $f : Q \to P$ such that

- $f(q_0) = p_0$,
- $q \in F_A \Leftrightarrow f(q) \in F_B$,
- and for every pair of transitions (q, φ, q') and $(f(q), \psi, f(q'))$, φ is satisfiable if and only if ψ is satisfiable.

Several algorithms for the minimization of SFA and an exhaustive explanation of the underlying theory can be found in [5]. We can attempt to extend the concept of a minimum to a PA A by applying it to the SFA A_μ which results from fixing a parameter assignment μ.

Definition A.4 (equivalence of states in PA)**.** Let $A = (M, Q, q_0, \delta, F)$ be a PA. Two states $q, p \in Q$ are equivalent, $q \sim p$, if $L^q(A_\mu) = L^p(A_\mu)$ for every parameter assignment $\mu \in \Theta$.

Definition A.5 (minimization of PA)**.** For any PA $A = (M, Q, q_0, \delta, F)$ that is deterministic per assignment, let $Min(A) = (M, Q_{/\sim}, q_{0/\sim}, \delta_{/\sim}, F_{/\sim})$ be the minimum of A, where

- $Q_{/\sim}$ is the quotient set of Q by $\sim$,
- $q_{0/\sim}$ is the equivalence class of q_0,
- $F_{/\sim}$ is the quotient set of F by $\sim$, i. e., the set of equivalence classes of states in F,
- and $\delta_{/\sim} = \{(q_{/\sim}, \bigvee_{\varphi \in S} \varphi, p_{/\sim}) \mid q_{/\sim}, p_{/\sim} \in Q_{/\sim}, S = \{\varphi \mid (q, \varphi, p) \in Q, q \in q_{/\sim}, p \in p_{/\sim}\}\}$.

It holds that $L(A) = L(Min(A))$.

If we want to see whether the isomorphism properties of minimized SFA hold up when applied to PA, we first need to define isomorphism for PA. Once more, there is no canonical or natural way to translate the concept to PA, since the behaviour of a PA is strongly influenced by its parameters. We choose a very strict definition of isomorphism that does not permit transformation of parameters aside from relabeling.

Definition A.6 (isomorphism of PA)**.** Let A and B be two PA, where Y_A is the set of parameters used by A and Y_B is the set of parameters used by B. Then A and B are isomorphic if there is a bijection $f : Y_B \to Y_A$ such that for every parameter assignment $\mu : Y_A \to D$, the SFA A_μ and $B_{\mu \circ f}$ are isomorphic.

This naturally implies $L(A) = L(B)$.

Theorem A.7. The minimum of PA is not unique up to isomorphism. There are PA A and B such that $L(A) = L(B)$, but $Min(A)$ is not isomorphic to $Min(B)$.

(a) A. (b) B.

Figure A.1: Two PA that are deterministic per assignment and describe the language L_3.

Proof. The example automata A and B both describe the language L_3. Both are deterministic per assignment and also their own minimum, because every PA that is deterministic per assignment and does not describe the universal or empty language needs to have at least two states. However, A and B are not isomorphic: the only possible bijection $f : Y_B \to Y_A$ maps z to y, and $L(A_\mu) \neq L(B_{\mu \circ f})$ for all $\mu : Y_A \to D$. For example, the word $(1, 1.5)$ is accepted by A_μ when $\mu(y) = 1$, but it is not accepted by $B_{\mu \circ f}$ since $1.5 > \mu(f(z)) = 1$. □

This result is discouraging at first, however it assumes that there is no known relation between A and B. We will see next that the product provides an interesting edge case where the minima of two distinct PA are isomorphic.

Proposition A.8 (minimization of product)**.** Let there be two PA $A = (M, Q, q_0, \delta_A, F_A)$ and $B = (M, P, p_0, \delta_B, F_B)$ that are deterministic per assignment. Then the PA $Min(A)$, $Min((A \times B, F_A \times P)$ and $Min((A \otimes B, F_A \times P)$ are isomorphic.

Proof. It is sufficient to prove that, for arbitrary $q \in Q, p \in P$, $L^q(A_\mu) = L^{(q,p)}((A \times B, F_A \times P)_{\mu'}) = L^{(q,p)}((A \otimes B, F_A \times P)_{\mu'})$ for any μ' where $\mu'|_{Y_A} = \mu$.

Let $w = (w_1, \ldots, w_n) \in L^{(q,p)}((A \times B, F_A \times P)_{\mu'})$ or $w = (w_1, \ldots, w_n) \in L^{(q,p)}((A \otimes B, F_A \times P)_{\mu'})$. Then in either case, there exists a sequence of states and transitions $((q, p), \varphi_1 \wedge \psi_1, (q_1, p_1)), \ldots, (q_{n-1}, p_{n-1}), \varphi_n \wedge \psi_n, (q_n, p_n))$ such that $q_n \in F_A$ and $\varphi_i \wedge \psi_i$ evaluates to true when plugging in w_i. Then the sequence $(q, \varphi_1, q_1), \ldots, (q_{n-1}, \varphi_n, q_n)$

is obviously a path of w in $A_{\mu'|_{Y_A}}$ that starts in q and terminates in an accepting state. Therefore, $w \in L^q(A_{\mu'|_{Y_A}})$.

For the other direction, let $w \in L^q(A_\mu)$ for some parameter assignment μ. Since A, $(A \times B, F_A \times P)$ and $(A \otimes B, F_A \times P)$ are deterministic per assignment, w will terminate a run in $(A \times B, F_A \times P)_{\mu'}$ and $(A \otimes B, F_A \times P)_{\mu'}$ in some accepting state in $F_A \times P$ for any parameter assignment μ' where $\mu'|_{Y_A} = \mu$. □