Models of Molecular Computing

Models of Molecular Computing

Proefschrift

ter verkrijging van
de graad van Doctor
aan de Universiteit Leiden,
op gezag van de Rector Magnificus Dr. D.D. Breimer,
hoogleraar in de faculteit der Wiskunde en
Natuurwetenschappen en die der Geneeskunde,
volgens besluit van het College voor Promoties
te verdedigen op woensdag 29 mei 2002
te klokke 16.15 uur

door

Nikè van Vugt
geboren te Waspik
in 1972

Promotiecommissie :

Promotor:	Prof. dr. G. Rozenberg
Co-promotor:	Dr. H.J. Hoogeboom
Referent:	Prof. dr. A. Salomaa (*Turun Yliopisto, Finland*)
Overige leden:	Prof. dr. G. Mauri (*Università degli Studi di Milano-Bicocca, Italië*)
	Dr. Gh. Păun (*Institute of Mathematics of the Romanian Academy, Roemenië*)
	Dr. L. Kari (*University of Western Ontario, Canada*)
	Dr. H.C.M. Kleijn
	Prof. dr. J.N. Kok
	Prof. dr. W.R. van Zwet

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

ISBN 90-77017-67-4

voor Jurriaan

Contents

1 Introduction **13**

2 Preliminaries **19**
- 2.1 Sets, words and languages 19
- 2.2 The Chomsky hierarchy 20
- 2.3 Operations on languages 23
- 2.4 Blind one-counter automata 25
- 2.5 Context-free valence grammars 27

I Splicing systems **29**

3 Definitions, examples and research topics **31**
- 3.1 Non-iterated splicing 32
- 3.2 Iterated splicing 34
- 3.3 Restricted non-iterated splicing 38
- 3.4 Research topics 40

4 String representations of splicing rules **41**
- 4.1 Families of splicing relations 42
  - 4.1.1 LIN and CF splicing rules 42
  - 4.1.2 FIN, REG, CS and RE splicing rules 43
- 4.2 Families of non-iterated splicing languages 44
- 4.3 Families of iterated splicing languages 46
- 4.4 Representations other than $\mathbb{Z}$ and $\not\Lambda$ 48
  - 4.4.1 Splicing with FIN, REG, CS or RE rules 48
  - 4.4.2 Splicing with LIN or CF rules 48
- 4.5 Summary 49

5 Non-iterated splicing with regular rules **51**
- 5.1 Unrestricted splicing 51
- 5.2 Same-length splicing 54
- 5.3 Self splicing 54

5.4 Length-decreasing splicing 57
5.5 Length-increasing splicing 58
5.6 Summary 62

6 Upper bounds for restricted non-iterated splicing 63
6.1 Same-length splicing 64
6.2 Splicing in *in* or *de* mode 65
6.3 Self splicing 67
6.4 Summary 71

II Sticker systems 73

7 Definitions, examples and research topics 75
7.1 Sticker systems 75
7.2 Sticker languages 77
7.3 Families of sticker languages 81
7.4 Research topics 84

8 Fair sticker languages 85
8.1 Fair sticker languages are BCA-languages 85
8.2 BCA-languages are codings of fair sticker languages 86
8.3 Summary 92

9 A hierarchy of sticker families 93
9.1 A primitive 'normal form' 93
9.2 Primitive computations 99
9.3 Fair computations 103
9.4 Primitive fair computations 104
9.5 Summary 106

III Forbidding and enforcing 107

10 Definitions, examples and research topics 109
10.1 Forbidding 109
10.2 Enforcing 111
10.2.1 Definitions and basic properties 111
10.2.2 Evolving through enforcing 112
10.2.3 A finitary normal form 113
10.3 Combining forbidding and enforcing 116
10.3.1 Definitions and examples 116
10.3.2 The structure of computation in fe systems 119
10.4 Research topics 120

11 Properties of forbidding sets and enforcing sets 123
11.1 Forbidding sets 123
11.1.1 Finite forbidding sets 123
11.1.2 Two useful normal forms 124
11.1.3 Minimal forbidding sets 126
11.1.4 Maximal forbidding sets 127
11.1.5 A tree representation for consistent families 128
11.2 Enforcing sets 128
11.2.1 Finite enforcing sets 128
11.2.2 Normal forms 129
11.2.3 Deterministic enforcing sets 129
11.3 Summary 131

12 Sequences of languages in forbidding-enforcing families 133
12.1 Converging sequences of languages 133
12.2 Forbidding-enforcing families are closed sets 134
12.3 Evolving sequences of languages 135
12.4 The importance of finite languages 136
12.5 Summary 138

Preface

This thesis comprises three parts, each of which discusses a model of (some aspects of) molecular computing: splicing systems, sticker systems, and forbidding-enforcing systems. All three models are rooted in formal language theory. The three parts are preceded by an introduction, which discusses the basic structure of DNA molecules as well as the origin of each of the models, and by preliminaries, which provide the main concepts of formal language theory that are used throughout the thesis.

Each part starts with a chapter that defines the model discussed in that part, gives examples and basic submodels, and ends with a description of the open problems that we intend to solve or the questions that we pose about certain aspects of the model. Each of the other chapters is based on a paper published already, or on a manuscript. Here is the list of these six papers and two manuscripts.

Splicing

(1) The power of H systems: does representation matter? [HV98]
(2) A characterization of non-iterated splicing with regular rules [DHV01]
(3) Upper bounds for restricted splicing [HV02]

Sticker systems

(4) Fair sticker languages [HV00]
($*$) A hierarchy of sticker families (manuscript, with H.J. Hoogeboom)

Forbidding and enforcing

(5) Forbidding and enforcing [EH$^+$00]
($\diamond$) Properties of forbidding sets and enforcing sets (manuscript, with A. Ehrenfeucht, H.J. Hoogeboom and G. Rozenberg)
(6) Sequences of languages in forbidding-enforcing families [EH$^+$01]

We describe now in more detail the correspondence between the chapters of this thesis and the papers and manuscripts listed above.

Chapter 4 is essentially paper (1); we have only changed the layout a little and added some explanations.

Chapters 5 and 6 result from a continuation of the research presented in paper (2). In Chapter 5 we present a characterization of the family of unrestricted non-iterated splicing languages generated by a linear initial language and a regular set of rules. Extending this result, we prove that under certain conditions regular rule sets may be replaced by finite rule sets, for unrestricted non-iterated splicing and for several cases of restricted non-iterated splicing. In Chapter 6 we determine upper bounds for the restricted splicing families that we discuss. This chapter is based on paper (3).

Chapter 8 contains the part of paper (4) where we answered the question whether fair sticker languages are context-free languages (or even linear languages) by proving that each fair sticker language is accepted by a blind one-counter automaton and showed that these blind one-counter languages constitute a rather close upper bound for the fair sticker languages.

In paper (4) we also proved a normal form for sticker systems: without changing the sticker language it is always possible to replace the complementarity relation by the identity. This normal form appears here in Chapter 7.

In Chapter 9, which corresponds to manuscript ($*$), we show that every (fair) sticker language can be generated by a sticker system that can do only primitive computations, provided that one is allowed to use a coding. Moreover, we compare the unrestricted sticker languages with the restricted versions of sticker languages that we consider.

Paper (5) is an overview of research done in the new field of forbidding-enforcing systems. It contains the results from the initial paper in that field ([ER]) as well as some results that we obtained when searching for standard formal language properties of the newly defined systems (like normal forms, determinism versus non-determinism and such), and when investigating the topological aspects of forbidding-enforcing families. The former research was described in manuscript ($\diamond$), the latter in paper (6). In Chapters 10, 11 and 12 we essentially give a considerably extended version of paper (5), where Chapter 11 corresponds to ($\diamond$) and Chapter 12 to (6).

During the years that I was a Ph.D. student, many people showed their sincere interest in my work, the progress I made with it and my personal well-being. Of all those people I especially wish to mention the entire Theoretical Computer Science group, which I found a very warm and friendly environment to work in (GR, Joost, Jetty, Hendrik Jan, Tjalling, Jurriaan, Maurice, Rudy, Sebastian, Pier and Marloes), as well as Henk, Frans, Tero, Walter, Jeannette, Siegfried, my family, especially Jurriaan, papa, mama, Kirsten, Godelief, Jeroen, tante Jo, oom Albert, Hans, Leida, Arjan, Ernst, and my landlady, mevrouw Vries.

I also gratefully acknowledge support by LIACS and TUCS, that made it possible for Jurriaan and me to work in Turku, Finland, during the rainy month of August 1998.

Chapter 1

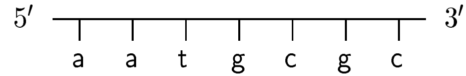
Introduction

DNA molecules and various operations on them can be conveniently expressed as strings and operations on strings. Hence, many models of DNA computation have been formulated within formal language theory. We consider here three formal language based models of molecular processes. Two of them, splicing systems and sticker systems, were defined during the last 15 years, while the third one is more recent: forbidding-enforcing systems.

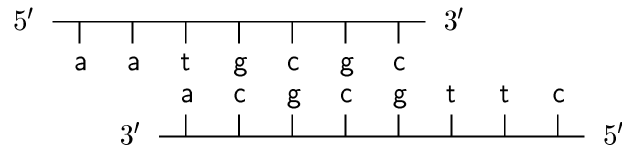
We will first describe the structure of DNA, as well as the properties of DNA that have motivated splicing and sticker systems. Then the three models considered in this thesis are (informally) introduced.

DNA

Our description of DNA (*deoxyribonucleic acid*) molecules and their manipulation is quite simplified, but adequate for this thesis. A DNA molecule (see, e.g., [PRS98]) is a chain of nucleotides. Each nucleotide consists of a sugar, a phosphate group and a base. Nucleotides can differ from each other only in their bases, which come in four sorts: *adenine, thymine, cytosine* and *guanine*, abbreviated by a, t, c and g, respectively. The sugar has five carbon atoms, numbered $1'$ through $5'$, which serve as 'attachment points' (referred to as 'ends'): the phosphate group is attached to the $5'$ end and the base to the $1'$ end. Two nucleotides can link through a bond between the (phosphate group at the) $5'$ end of one nucleotide and the $3'$ end of the other. In this way, an alternating sequence of sugars and phosphate groups forms the backbone of a DNA molecule, and this backbone has an orientation: on one end there is a free $3'$ end and on the other end there is a free $5'$ end. A chain of nucleotides formed in this way is called a *single stranded* DNA *molecule* or simply a *strand*, and is identified by the order in which the bases appear on the backbone (usually read from the $5'$ end to the $3'$ end). Schematically such a single strand of DNA may be represented as follows.



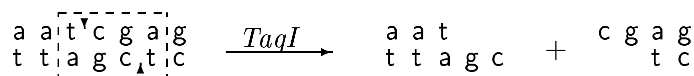
An important feature of DNA is the so-called *Watson-Crick complementarity*, which is the phenomenon that the four bases actually come in two pairs: a pairs with t, and c pairs with g, meaning that a and t can form a (weak) hydrogen bond, and the same holds for c and g; moreover no other pairs can form a hydrogen bond. This Watson-Crick complementarity allows two single strands to form a *double stranded* DNA molecule: the two strands bind together ('anneal') through hydrogen bonds between complementary bases positioned opposite to each other in the two strands. For example, if in a solution containing the single stranded molecule aatgcgc depicted above there is also a single stranded molecule cttgcgca (also read from the $5'$ end to the $3'$ end), they may together form the partially double stranded molecule shown below.

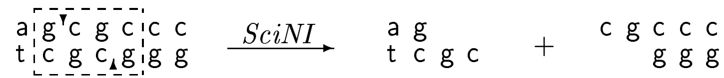


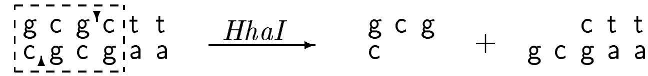
This figure illustrates also the second necessary condition for two single strands to form a double strand: the orientation of one strand is opposite to the orientation of the other. The single stranded pieces of DNA at the ends of the partially double stranded molecule are referred to as *sticky ends*, because they allow DNA molecules to stick together to form longer molecules. Thus, e.g., if there is another molecule in the solution beginning with aag or tt (at its $5'$ end), this molecule can anneal with the structure above. This annealing of single strands to form double strands, or of (partially) double strands to form larger (partially) double strands is spontaneous, and is referred to as *self-assembly*.

Usually, in the illustrations the backbones are not depicted, and by convention the upper strand is written in the $5'$ to $3'$ direction (hence the lower strand in the $3'$ to $5'$ direction).

Double stranded DNA molecules can be cut in two partially double stranded pieces by *restriction enzymes*, which look for a specific recognition site and then cut the molecule somewhere within (or sometimes outside) this recognition site. We show this for the three restriction enzymes *TaqI*, *SciNI* and *HhaI*, using three sample molecules. The dashed boxes denote the recognition sites, and the small black triangles denote the cutting points.







Since *TaqI* and *SciNI* leave exactly the same sticky ends – i.e., both the base sequences and the orientations of the overhangs are the same – the two molecules above that are cut by these two enzymes can recombine into two new molecules (or the two original molecules can be restored):

Sticky ends produced by *HhaI*, however, have a different orientation, hence molecules that are created through cleavage by *HhaI* cannot be combined in this way with molecules resulting from cutting by *TaqI* or *SciNI*.

The cutting by restriction enzymes and subsequent recombination into old and new molecules described above is also called *splicing*.

Abstracting from their biochemical properties, single DNA strands can be seen as strings over the alphabet $\{\mathsf{a}, \mathsf{c}, \mathsf{g}, \mathsf{t}\}$. Fully double stranded molecules can then be represented by two strings of which one is written on top of the other, as we did above, but in our notation we can also leave one of the strands out, because the other one can be deduced using the Watson-Crick complementarity. Since in this way molecules can be represented as strings, operations on molecules, such as self-assembly or splicing, can be described through operations on strings – this brings us into the framework of formal language theory.

Splicing systems

Splicing systems are designed to model the cutting and recombination of DNA molecules in the presence of restriction enzymes ([Hea87]). Originally, splicing systems were defined in such a way that the natural process of cutting and recombination was described as accurately as possible: a finite set of initial strings representing the initial molecules could be spliced according to two finite sets of splicing rules, in which each rule represented the recognition site and cutting points of one restriction enzyme as a 3-tuple (u, x, v), where uxv is the recognition site and x is the overhang left after cutting (hence *TaqI* is represented as $(\mathsf{t}, \mathsf{cg}, \mathsf{a})$). One of these two sets consisted of splicing rules that leave a $5'$ overhang after cutting, the other one of splicing rules leaving

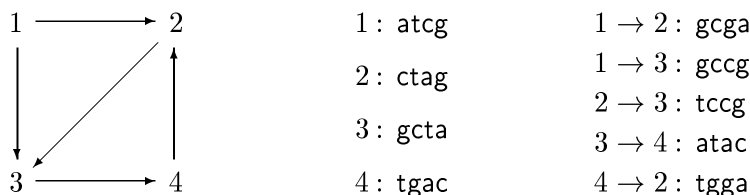
a $3'$ overhang. Two initial strings $u'uxvv'$ and $y'ywzz'$ could be spliced only by using two rules (u, x, v) and (y, w, z) from the same set that left the same overhang, i.e., $x = w$. The resulting strings, apart from $u'uxvv'$ and $y'yxzz'$ themselves, would then be $u'uxzz'$ and $y'yxvv'$.

Later, two rules (u, x, v) and (y, x, z) representing compatible enzymes were combined into a 4-tuple (ux, v, yx, z) or even into a string $ux\#v\$yx\#z$ (both also called a splicing rule). Here the $ symbol separates the recognition sites and the # symbols indicate the cutting points. As the result of splicing $u'uxvv'$ and $y'yxzz'$ using $ux\#v\$yx\#z$ only $u'uxzz'$ was considered, because one could easily obtain also $y'yxvv'$ by adding the symmetric rule $yx\#z\$ux\#v$.

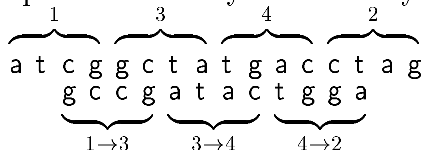
Finally, several aspects in the definition of splicing systems were generalised: a splicing rule $u_1\#u_2\$u_3\#u_4$ was no longer required to represent two enzymes (let alone two compatible enzymes), the alphabet was no longer restricted to $\{\mathsf{a}, \mathsf{t}, \mathsf{c}, \mathsf{g}\}$, it was no longer required that for each rule $u_1\#u_2\$u_3\#u_4$ also the symmetric rule $u_3\#u_4\$u_1\#u_2$ be present, and the set of initial strings and the set of rules no longer had to be finite. All these abstractions lead to a model of splicing based on formal language theory, in which the effect of splicing sets of initial strings using sets of splicing rules can be investigated. In particular, the power of the splicing operation is studied as a function of the complexity of the set of initial strings and the set of rules.

Sticker systems

Adleman ([Adl94]) uses the Watson-Crick complementarity to propose a biochemical implementation of an algorithm to solve the Hamiltonian Path Problem: the question whether a given graph contains a path going through each of its nodes exactly once (starting from a designated initial node, and ending in a designated terminal node). In Adleman's scheme nodes are represented by short DNA strands, and edges are designed to match with the second half of their source node and the first half of their target node. We illustrate this for a very small example.



Now, when the strands representing the nodes and edges are placed in a solution, paths in the graph are formed by self-assembly.



After that the Hamiltonian paths may be detected in the solution by a rather involved biochemical selection process.

Sticker systems are introduced as a model for the self-assembly phase of Adleman's experiment ([KP+98]). A sticker system specifies finite sets of upper and lower 'stickers' (single stranded molecules), and a finite set of axioms (used as a seed for the process joining upper and lower strands). The complementarity relation is modelled by a binary relation on the alphabet. Roughly speaking, the language generated by the system consists of all strings formed by upper stickers for which an exactly matching (i.e., complementary) sequence of lower stickers can be found.

These sticker systems are generalised to sticker systems that have (partially) double stranded stickers, or axioms to which both to the right and to the left stickers can be attached, etcetera. Furthermore, motivated by the wish to find a model that is computationally complete, in addition to the unrestricted computations described above, restricted computations are considered: for example, computations that are only valid if the number of upper stickers used equals the number of lower stickers used.

The theory of sticker systems investigates the relationships between different types of sticker systems as well as their relationship to various types of grammars and automata.

Several other models of the use of self-assembly for computations are considered in the literature, see, e.g., [WYS98] and [RW+98].

Forbidding-enforcing systems

In a completely different kind of molecular (not necessarily DNA) model *boundary conditions* are used to describe what can happen in a molecular system. We consider here two types of boundary conditions: forbidding and enforcing. *Forbidding conditions* say that if a certain group of components (i.e., parts of molecules) is present in the system, then the system will lose its functionality (e.g., an organism will die, or a molecular computation will go 'the wrong way') – hence such a combination of components is forbidden. *Enforcing conditions* say that if a certain group of molecules is present in the system, then (as the result of a molecular reaction) some other molecules will eventually be present in the system. Hence the evolution of a system described by forbidding conditions $\mathcal{F}$ and enforcing conditions $\mathcal{E}$ will proceed according to the reactions described by $\mathcal{E}$ but restricted in such a way that none of the forbidden combinations from $\mathcal{F}$ will be created.

Such forbidding-enforcing systems, *fe systems* for short, are more 'tolerant' in describing results of (molecular) computations than the standard grammatical models: one fe system describes a whole family of outcomes all of which obey the forbidding and enforcing constraints of the system. Thus in the case that we model the molecules by strings (as we do in this thesis), one fe system

specifies a possibly infinite family of languages. A language belongs to this family if and only if it is consistent with the forbidding conditions and satisfies the enforcing conditions – nothing else is required from the language. Intuitively speaking, fe systems follow the rule "everything that is not forbidden is allowed", while standard formal language theory (grammars and automata) follows the dual rule "everything that is not allowed is forbidden".

Since the forbidding and enforcing conditions can be expressed by strings and languages, we get again a formal language theoretic model (albeit nonstandard).

Chapter 2

Preliminaries

In order to fix our notation, we recall some well-known formal language theory concepts that we need throughout this thesis. More details can be found in, e.g., [HU79], [RS97].

2.1 Sets, words and languages

The set of positive natural numbers is denoted by $\mathbb{N}$, and the set of integers by $\mathbb{Z}$. We write $\mathbb{Z}^k$, for some $k \geq 1$, for the set $\{(v_1, \ldots, v_k) \mid v_1, \ldots, v_k \in \mathbb{Z}\}$, and similarly A^k for an arbitrary set A. Elements of $\mathbb{Z}^k$, for any $k \geq 1$, are also written as $\vec{v}$, and the vector consisting of k 0's is written as $\vec{0}$. We denote the largest element of a finite subset A of $\mathbb{Z}$ by $\max A$.

For two sets A and B, $A \subseteq B$ denotes the inclusion of A in B, and $A \subset B$ denotes the proper inclusion of A in B (i.e., additionally, $A \neq B$). By $A - B$ we denote the set consisting of all elements of A that are not elements of B. We use $\mathcal{P}(A)$ to denote the set consisting of all subsets of A, called the power set of A. We denote the empty set by $\varnothing$, and the cardinality of a set A by $\#(A)$. We often notationally identify a singleton set with its element.

An alphabet is a finite set of symbols (letters), and a word (string) over an alphabet Σ is a finite sequence of letters from Σ. We denote the empty word by λ, the length of a word w by $|w|$, and the number of occurrences of a symbol a in w by $\#_a(w)$. The concatenation of two words x and y is denoted by $x \cdot y$ or simply xy.

A language over Σ is a (possibly infinite) set of words over Σ. The language consisting of all words over Σ is denoted by Σ^*, and Σ^+ denotes the language $\Sigma^* - \{\lambda\}$. A set of languages containing at least one language not equal to $\varnothing$ or $\{\lambda\}$ is also called a family of languages. For a language K and an integer $n \geq 0$, $K|_{\leq n} = \{w \in K \mid |w| \leq n\}$, and $K|_{>n} = \{w \in K \mid |w| > n\}$. A sequence of languages $K_1, K_2, \ldots$ for which $K_1 \subseteq K_2 \subseteq \ldots$ is called ascending.

The set of prefixes of a given word $w \in \Sigma^*$ is defined as $\mathrm{Pref}\,(w) = \{u \in$

$\Sigma^* \mid w = uv$ for a $v \in \Sigma^*\}$, and the set of suffixes as $\mathrm{Suf}(w) = \{v \in \Sigma^* \mid w = uv$ for a $u \in \Sigma^*\}$.

The following notions are frequently used in Part III. A word $x \in \Sigma^*$ is a subword of a word $y \in \Sigma^*$, denoted $x \ \underline{\mathrm{sub}}\ y$, if $y = uxv$ for some $u, v \in \Sigma^*$. We say that x is a subword of a language K if $x \ \underline{\mathrm{sub}}\ y$ for some $y \in K$. The set of subwords of x is denoted by $\underline{\mathrm{sub}}\,(x)$, and the set of subwords of K by $\underline{\mathrm{sub}}\,(K)$, hence $\underline{\mathrm{sub}}\,(K) = \bigcup_{x \in K} \underline{\mathrm{sub}}\,(x)$. Note that, for two languages K and L, $K \subseteq \underline{\mathrm{sub}}\,(L)$ if and only if $\underline{\mathrm{sub}}\,(K) \subseteq \underline{\mathrm{sub}}\,(L)$. A language K is called subword free if, for all $x, y \in K$, $x \ \underline{\mathrm{sub}}\ y$ implies $x = y$.

For a language K we define $\underline{\mathrm{sub}}_{\mathrm{max}}(K)$, the set of maximal subwords of K, to be $\{x \in K \mid$ there is no $y \in K$ with $x \neq y$ and $x \ \underline{\mathrm{sub}}\ y\}$. Obviously, for every subword free language K we have $K = \underline{\mathrm{sub}}_{\mathrm{max}}(K)$. Furthermore, note that for infinite K it may be that $\underline{\mathrm{sub}}_{\mathrm{max}}(K)$ conveys little information concerning K, since for instance $\underline{\mathrm{sub}}_{\mathrm{max}}(a^+) = \varnothing$.

From the definitions it is clear that $\underline{\mathrm{sub}}_{\mathrm{max}}(K) \subseteq K \subseteq \underline{\mathrm{sub}}\,(K)$ for any language K. Moreover, for a finite or subword free language K it holds that $\underline{\mathrm{sub}}\,(\underline{\mathrm{sub}}_{\mathrm{max}}(K)) = \underline{\mathrm{sub}}\,(K)$. This directly implies the following property: if K and L are finite or subword free languages, then $\underline{\mathrm{sub}}\,(K) = \underline{\mathrm{sub}}\,(L)$ if and only if $\underline{\mathrm{sub}}_{\mathrm{max}}(K) = \underline{\mathrm{sub}}_{\mathrm{max}}(L)$. Furthermore, this property and the fact that for subword free K it holds that $K = \underline{\mathrm{sub}}_{\mathrm{max}}(K)$ together prove the following lemma.

Lemma 2.1 *Let K and L be subword free languages.*
Then $\underline{sub}\,(K) = \underline{sub}\,(L)$ if and only if $K = L$.

2.2 The Chomsky hierarchy

The family of all finite languages is denoted by FIN.

A *deterministic finite automaton* (DFA) is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is the input alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, q_0 is the initial state and $F \subseteq Q$ is the set of final states.

We extend δ to a mapping from $Q \times \Sigma^*$ to Q by defining $\delta(p, \lambda) = p$ and $\delta(p, aw) = \delta(\delta(p, a), w)$, for $p \in Q$, $a \in \Sigma$ and $w \in \Sigma^*$.

A triple $(w, p, z) \in \Sigma^* \times Q \times \Sigma^*$, called an instantaneous description, describes the current situation of the automaton: wz is the input word, of which w is already read and z still has to be read, and p is the current state. When the automaton has a transition (p, a, q), i.e., $\delta(p, a) = q$, then an instantaneous description (w, p, az) can change into (wa, q, z), denoted $(w, p, az) \vdash (wa, q, z)$. The reflexive and transitive closure of $\vdash$ is denoted $\vdash^*$. The language accepted by the finite automaton $\mathcal{A}$ is defined as $L(\mathcal{A}) = \{w \in \Sigma^* \mid (\lambda, q_0, w) \vdash^* (w, f, \lambda)$ for a final state $f \in F\}$, and is called a *regular language*.

We use the notation REG for the family of regular languages.

Useful variants of the DFA are: the *non-deterministic* finite automaton, that has a transition relation rather than a transition function, i.e., $\delta \subseteq Q \times \Sigma \times Q$; the finite automaton that may have λ-transitions (i.e., transitions of the form (p, λ, q)); and the *lazy* finite automaton, that allows transitions of the form (p, w, q), where $w \in \Sigma^*$. All these variants are known to be equivalent to the DFA, where two language-generating or language-accepting devices are called equivalent if they define the same language.

A finite automaton is usually represented graphically: the states are indicated by circles with the name of the state written in it, and a transition (p, a, q) is represented by an arrow from the circle containing p to the circle containing q, labelled by a. The initial state is indicated with a '$\Rightarrow$' symbol and the final states by two concentric circles.

A *context-free grammar* (CFG) is a 4-tuple $G = (N, T, P, S)$, where N is a finite set of non-terminals, T is a finite set of terminals, $P \subseteq N \times (N \cup T)^*$ is a finite set of productions and $S \in N$ is the start symbol. A production (A, α) with $A \in N$ and $\alpha \in (N \cup T)^*$ is written as $A \to \alpha$. If α equals λ, the production is called a λ-production.

A string x is said to derive a string y in G, denoted $x \Rightarrow_G y$ (or $x \Rightarrow y$ if G is clear from the context), if $x = wAz$ and $y = w\alpha z$, for some $w, z \in (N \cup T)^*$, and there is a production $A \to \alpha$ in P. The reflexive and transitive closure of $\Rightarrow$ is denoted $\Rightarrow^*$. A string $x \in (N \cup T)^*$ with $S \Rightarrow^* x$ is called a sentential form of G. The *context-free language* generated by G is defined as $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$.

The family of all context-free languages is denoted by CF.

A *linear* context-free grammar is a CFG in which each production has at most one non-terminal in its right-hand side, i.e., every production is of the form $X \to wYz$ or $X \to w$, with $X, Y \in N$ and $w, z \in T^*$. The language generated by such a grammar is called a *linear (context-free) language*. We use LIN to denote the family of all linear languages.

A CFG is called *regular* (or right-linear) if each production is of the form $X \to wY$ or $X \to w$, with X and Y non-terminals and $w \in T^*$. It is called regular because its derivations correspond to paths from the initial to a final state in a lazy finite automaton. Indeed, the regular CFG's generate exactly the regular languages.

The machine counterpart of the context-free grammar is the *pushdown automaton* (PDA). A PDA is essentially a finite automaton with an external storage device: a stack. Depending on the current state and input symbol and on the top symbol on the stack, the PDA moves to another state and replaces the top stack symbol by a finite number of stack symbols.

A *linear bounded automaton* (LBA) is an extension of a finite automaton, in the sense that it may move back and forth on its input and even overwrite (input) symbols with other symbols. It is called linear bounded because it is

not allowed to use more of the input tape than the part that contains the input string. For that purpose the start and end of the input string are marked with two (different) special symbols.

Formally, an LBA is an 8-tuple $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, \triangleleft, \triangleright, q_0, F)$, where Q is the finite set of states, $\Sigma \subseteq \Gamma$ is the input alphabet, Γ is the tape alphabet, $\triangleleft, \triangleright \in \Gamma$, with $\triangleleft \neq \triangleright$, are the left and right endmarkers, respectively, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. The transition relation δ is a finite subset of $Q \times \Gamma \times Q \times \Gamma \times \{L, R, N\}$. An element (p, a, q, b, r) of δ is interpreted as follows: when $\mathcal{A}$ is in state p and reads the symbol a it may change its state to q, overwrite a with b and go to the previous (if $r = L$) or the next (if $r = R$) symbol on the tape, or stay at the same position (if $r = N$). Furthermore, the following requirements must be satisfied: $a = \triangleleft$ if and only if $b = \triangleleft$; if $a = \triangleleft$ then $r \neq L$; $a = \triangleright$ if and only if $b = \triangleright$; if $a = \triangleright$ then $r \neq R$.

The language accepted by an LBA consists of all input words for which, when starting in q_0, the LBA can reach a final state. It is called a *context-sensitive language*, and the family of context-sensitive languages is denoted by CS.

An LBA can simulate the productions of a CFG on a 'second track' of its input tape, thereby providing a proof that $\mathsf{CF} \subseteq \mathsf{CS}$. This second track is 'created' by using symbols consisting of two components, of which the first is the old symbol and the second is the symbol on the corresponding position of the second track. The LBA writes the start symbol of the CFG under consideration on its second track and then repeatedly chooses one of the non-terminals on the second track and rewrites it according to an applicable production, while shifting the symbols on the second track if necessary. Such a simulation of a derivation of a word w by a CFG can be done within the part of the input tape that contains w, because if a CFG does not have λ-productions, then the sentential forms in the derivation of w by this CFG never need to be longer than w, and for every CFG with λ-productions an equivalent (modulo λ) CFG without λ-productions can be constructed.

At the top of the automata hierarchy we have the *Turing machine* (TM). It is a generalisation of the LBA in the sense that it is allowed to use an infinitely long input tape to carry out its computations, instead of only the part of the tape where the input word is.

Formally, a TM is a construct $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where Q, Σ, Γ, q_0 and F are as for the LBA, $Q \cap \Gamma = \varnothing$, δ is a subset of $Q \times \Gamma \times Q \times \Gamma \times \{L, R, N\}$, and $B \in \Gamma$ is the blank symbol, representing an empty cell on the tape. The current situation on the tape of a TM is described by an instantaneous description xqy with $x, y \in \Gamma^*$ and $q \in Q$. Here x contains the contents of the tape immediately to the left of the head, starting with the leftmost symbol that is not B, the head is on the first symbol of y, and y contains everything to the right of the head, up to and including the rightmost symbol that is not a blank. The language accepted by a TM is defined similar to the language accepted by an LBA, and is

called a *recursively enumerable* language. The family of recursively enumerable languages is denoted by RE.

The families FIN, REG, LIN, CF, CS and RE are said to form the Chomsky hierarchy. It is indeed a hierarchy because FIN $\subset$ REG $\subset$ LIN $\subset$ CF $\subset$ CS $\subset$ RE.

2.3 Operations on languages

A homomorphism is a function from Δ to Σ^*, for alphabets Δ and Σ, assigning to each symbol in Δ a string over Σ. It is called λ-free if it maps symbols in Δ to non-empty strings over Σ. A homomorphism h can be extended to a function from Δ^* to Σ^* as follows: $h(\lambda) = \lambda$, and $h(ax) = h(a)h(x)$, for $a \in \Delta$ and $x \in \Delta^*$.

A letter to letter homomorphism is called a coding. For a language family $\mathcal{F}$, we use COD($\mathcal{F}$) to denote the family of codings of languages in $\mathcal{F}$, where the coding h of a language L is defined as $h(L) = \{h(x) \mid x \in L\}$.

For a language L and a string w we define the left-quotient of L by w as the set $\{\ x \mid wx \in L\ \}$. The (right-)quotient of two languages L_1 and L_2, denoted L_1/L_2, is defined as the set $\{x \mid \text{there exists } y \text{ in } L_2 \text{ such that } xy \text{ is in } L_1\}$.

The shuffle of a language $K \subseteq \Delta^*$ with a symbol c is defined as $\{ucv \mid uv \in K \text{ for some } u, v \in \Delta^*\}$, for an alphabet Δ. Substitution with regular sets is defined by a mapping from Δ into regular subsets of Σ^*, for alphabets Δ and Σ.

A *finite-state transducer* (FST) $\gamma = (Q, \Sigma, \Delta, \delta, q_0, F)$ is a non-deterministic finite-state automaton with additional output, i.e., Q, Σ, q_0 and F are as defined for finite automata, Δ is the output alphabet, and δ is a finite set of transitions of the form $(p, a, w, q) \in Q \times (\Sigma \cup \{\lambda\}) \times \Delta^* \times Q$. Using such a transition, the machine may change from state p into state q, while reading a on its input and writing the string w to its output. In the literature such a transducer is also called 'a-transducer' or 'rational transducer'. The FST defines a relation in $\Sigma^* \times \Delta^*$, called an FST mapping or a finite-state transduction. If the transition relation of the FST is in $Q \times (\Sigma \cup \{\lambda\}) \times \Delta^+ \times Q$, then the corresponding FST mapping is called non-erasing.

A finite-state transducer that is not allowed to read λ (and at the same time write a (non-empty) string to the output), i.e., an FST with transition relation $\delta \subseteq Q \times \Sigma \times \Delta^* \times Q$, is called a *generalised sequential machine* (GSM).

Note that, for instance, homomorphisms, intersection with regular sets, and quotient with symbols or regular sets are special cases of the GSM mapping.

The effect of an FST mapping can also be achieved by the combination of an (arbitrary) inverse homomorphism, intersection with a regular set and a homomorphism. In the case of a non-erasing FST mapping the homomorphism may be assumed to be λ-free. The inverse homomorphism is used to 'guess' a transition (p, a, w, q) of the FST for each symbol a in the input, the intersection

with a regular set guarantees that the sequence of transitions corresponding to the input word obtained in this way forms indeed a path from the initial state to a final state of the FST, and the homomorphism then translates each transition (p, a, w, q) in this path to w, thus forming the output.

A family of languages $\mathcal{F}$ is said to be closed under a given n-ary operation on languages if whenever this operation is applied to n languages from $\mathcal{F}$, the result is again a language in $\mathcal{F}$. We summarize some well-known closure properties of the families in the Chomsky hierarchy in Table 2.1. Here we use '+' to denote closure and '−' to denote non-closure of a family under an operation.

	FIN	REG	LIN	CF	CS	RE
concatenation with symbols	+	+	+	+	+	+
concatenation	+	+	−	+	+	+
non-erasing FST mapping	−	+	+	+	+	+
FST mapping	−	+	+	+	−	+
non-erasing GSM mapping	+	+	+	+	+	+
GSM mapping	+	+	+	+	−	+
λ-free homomorphism	+	+	+	+	+	+
homomorphism	+	+	+	+	−	+
inverse homomorphism	−	+	+	+	+	+
intersection with regular sets	+	+	+	+	+	+
quotient with symbols	+	+	+	+	+	+
shuffle with symbols	+	+	+	+	+	+
substitution with regular sets	−	+	+	+	−	+
union	+	+	+	+	+	+

Table 2.1: Some closure properties of the Chomsky families

A family of languages that is closed under λ-free homomorphisms, inverse homomorphisms and intersection with regular languages is also called a trio; equivalently, a trio is closed under non-erasing FST mappings. When the λ-free homomorphisms may be replaced by arbitrary homomorphisms, the family is called a full trio. A trio is also sometimes called a faithful rational cone, while a full trio is called a rational cone. The Chomsky families REG, LIN, CF and RE are full trios, and CS is a trio. A (full) trio that is additionally closed under union is called a (full) semi-AFL, where AFL is an acronym for Abstract Family of Languages. The theory of AFL's and AFA's (Abstract Families of Automata) concerns itself with common properties of families of languages (see, e.g., [GG69, Gin75]), and the relation between properties of automata and properties of the families of languages they define.

2.4 Blind one-counter automata

The following type of automaton will be used in Part II.

A *blind one-counter automaton* (BCA) is a finite-state device equipped with an external memory (the 'counter') that contains an integer value which may be incremented and decremented by the automaton. It is called blind because the automaton cannot test its counter value during the computation, i.e., it may not check whether its counter value is zero and act according to the outcome of this test.

Formally, a BCA is a 5-tuple $\mathcal{B} = (Q, \Sigma, \delta, q_0, F)$, where Q, Σ, q_0 and F are as for finite automata, and $\delta \subseteq Q \times \Sigma \times \{-1, 0, 1\} \times Q$ is a finite set of instructions (or transitions).

An instantaneous description of $\mathcal{B}$ is an element of $Q \times \Sigma^* \times \mathbb{Z}$. For two instantaneous descriptions (p, ax, i) and (q, x, j), we write $(p, ax, i) \vdash (q, x, j)$ if $(p, a, \varepsilon, q) \in \delta$ and $j = i + \varepsilon$. By $\vdash^*$ we denote the reflexive and transitive closure of $\vdash$.

The *(blind one-counter) language accepted by* $\mathcal{B}$ consists of all strings for which the automaton in a computation on this string ends in a final state *and* at the same time has counter value zero. It is defined as $L(\mathcal{B}) = \{x \in \Sigma^* \mid (q_0, x, 0) \vdash^* (f, \lambda, 0) \text{ for some } f \in F\}$. The family of all languages accepted by blind one-counter automata (BCA-languages) is called $\mathsf{1BCA}$.

The blind one-counter automaton can be 'implemented' on a more commonly known device: the stack of a pushdown automaton may act as a counter. Consequently $\mathsf{1BCA} \subseteq \mathsf{CF}$. Since the context-free language $\{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w) \text{ and } \#_a(x) \geq \#_b(x) \text{ for every prefix } x \text{ of } w\}$ is not in $\mathsf{1BCA}$ (see [Gre78, Theorem 3]), we even have $\mathsf{1BCA} \subset \mathsf{CF}$.

In contrast with the definition of BCA given in [Gre78], we do not allow λ-instructions, i.e., instructions of the form $(p, \lambda, \varepsilon, q)$. However, these two definitions are equivalent, which can be explained as follows.

From AFA/AFL theory it is known that if there is a language L that describes the 'acceptance behaviour' of a certain type of automaton that has an additional storage, then the family of languages generated by this kind of automaton *without* λ-instructions equals the faithful rational cone generated by L, while the family of languages generated by this kind of automaton *with* λ-instructions equals the rational cone generated by L. For BCA's it is clear that the possible successful instruction sequences are naturally modelled by the 'two-sided Dyck language' $D_1^* = \{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w)\}$, where a and b represent addition of $+1$ and -1, respectively. Hence $\mathsf{1BCA}$ equals $\mathcal{C}^f(D_1^*)$, the faithful rational cone generated by D_1^*, and the family of languages generated by BCA's that can have λ-instructions is equal to $\mathcal{C}(D_1^*)$, the rational cone generated by D_1^*. In [Lat79, Proposition II.11] it is proved (as a special case of a more general result) that $\mathcal{C}^f(D_1^*) = \mathcal{C}(D_1^*)$. Hence the BCA's with λ-instructions are equivalent to the BCA's without λ-instructions.

We give now a more detailed explanation of the equivalence of 1BCA and the faithful rational cone generated by D_1^* (i.e., we explain the idea behind the AFA/AFL result used above). A BCA $\mathcal{B}$ can be seen as a finite-state device γ mapping input strings to strings over $\{a, b\}$ according to the instructions executed during the computation. The input is accepted precisely when the output belongs to D_1^*. Hence for each transition (p, c, ϵ, q) of $\mathcal{B}$, γ has a transition (p, c, α, q), where $\alpha = a$ if $\epsilon = +1$, $\alpha = b$ if $\epsilon = -1$ and $\alpha = \lambda$ if $\epsilon = 0$. Furthermore, $c \neq \lambda$ since $\mathcal{B}$ does not have λ-transitions, hence γ is a GSM (an arbitrary one, since α may be λ) with an extra acceptance criterium. Clearly $x \in L(\mathcal{B})$ if and only if $\gamma(x) \cap D_1^* \neq \varnothing$.

Now let a finite-state device τ be constructed from γ by giving τ a transition (p, α, c, q) for each transition (p, c, α, q) of γ. Then τ is a non-erasing finite-state transducer, because α may be λ but c may not. Moreover, since the only difference between γ and τ is that the roles of input and output have been interchanged in every transition (the structure of the underlying automaton is the same), it holds that $\gamma(x) \cap D_1^* \neq \varnothing$ if and only if $x \in \tau(D_1^*)$. Consequently $1\mathsf{BCA} \subseteq \{\tau(D_1^*) \mid \tau \text{ is a non-erasing finite-state transducer}\}$. Conversely, every non-erasing FST with input alphabet $\{a, b\}$ can be converted into a BCA without λ-transitions, following a procedure similar to the one described above. Hence $1\mathsf{BCA} = \{\tau(D_1^*) \mid \tau \text{ is a non-erasing FST}\}$, and according to Corollary 1 to Theorem 3.2.1 from [Gin75], the latter family equals the smallest trio containing D_1^*, which is also called the faithful rational cone generated by D_1^*. In other words, the family 1BCA is equal to the smallest language family that contains D_1^* and is closed under λ-free homomorphisms, inverse homomorphisms and intersection with regular languages.

The equivalence of the family of languages generated by BCA's that can have λ-instructions and $\mathcal{C}(D_1^*)$, the rational cone generated by D_1^*, can be shown in a similar way (but now γ and τ are arbitrary FST's instead of a GSM and a non-erasing FST, respectively, and the λ-free homomorphisms above are replaced by arbitrary homomorphisms).

From the discussion above we conclude that 1BCA is a principal rational cone (i.e., a rational cone generated by a single language), and in particular that 1BCA is closed under codings (being a special case of a homomorphism), left-quotient with strings (which can be done by a GSM), and union (using a construction similar to the one proving closure of REG under union). Because of the latter property 1BCA is also a full principal semi-AFL.

Blind one-counter automata were also studied as 'integer weighted finite automata' in [HH99] and as 'additive regular valence grammars (over $\mathbb{Z}$)' in [Pău80] (see also [FS97]). In these devices the instructions (productions) are assigned an integer value, and one considers only computations (derivations) for which these values add to 0.

In the next section we consider a generalisation of the BCA, the context-free valence grammar over $\mathbb{Z}^k$, that we need in Chapter 6. Both these concepts,

BCA's and valence grammars, are also investigated in [Hoo02].

2.5 Context-free valence grammars

A context-free *valence grammar over* $\mathbb{Z}^k$, for some $k \geq 0$, is a context-free grammar in which each production has a vector from $\mathbb{Z}^k$ associated to it. Derivations of context-free valence grammars are defined exactly as for normal context-free grammars, but now a derivation is only valid if the (componentwise) sum of the valences of the used productions is $\vec{0}$.

Formally, a context-free valence grammar (over $\mathbb{Z}^k$) is a 4-tuple $G = (N, T, R, S)$ with N, T and S as for a normal CFG, and $R \subseteq N \times (N \cup T)^* \times \mathbb{Z}^k$. An element $(A, \alpha, \vec{r})$ of R is written as $(A \to \alpha, \vec{r})$. $A \to \alpha$ is the underlying production and $\vec{r}$ is the valence of the production.

The *context-free valence language over* $\mathbb{Z}^k$ generated by G is defined as $L(G) = \{w \in T^* \mid (S, \vec{0}) \Rightarrow^* (w, \vec{0})\}$, where $(wAz, \vec{v}) \Rightarrow (w\alpha z, \vec{s})$ if and only if there is a production $A \to \alpha$ with valence $\vec{r}$ in R and $\vec{s} = \vec{v} + \vec{r}$.

For each $k \geq 0$, we denote the family of context-free valence languages over $\mathbb{Z}^k$ by $\mathsf{CF}(\mathbb{Z}^k)$. We will sometimes abbreviate the term 'context-free valence grammar (language) over $\mathbb{Z}^k$' to simply 'valence grammar (language)'.

As an aside, note that blind counter automata with k counters, for $k \geq 0$, can also be seen as regular valence grammars over $\mathbb{Z}^k$.

Clearly, a context-free valence grammar over $\mathbb{Z}^0$ is a normal context-free grammar, i.e., $\mathsf{CF} = \mathsf{CF}(\mathbb{Z}^0)$. Moreover, we have $\mathsf{CF}(\mathbb{Z}^k) \subseteq \mathsf{CF}(\mathbb{Z}^{k+1})$ for each $k \geq 0$.

Similar to the case of LBA's simulating derivations of CFG's, an LBA can simulate the derivations of a given context-free valence grammar over $\mathbb{Z}^k$, for some $k \geq 0$. For that we use the fact that for every valence grammar an equivalent (modulo λ) valence grammar in Chomsky normal form can be constructed, which means that each production in the underlying CFG is of the form $A \to BC$ or $A \to a$, for A, B, C non-terminals and a a terminal symbol (see [FS00, Theorem 5.1]; actually, in that theorem there are also restrictions on the valences, but those are not important to us here). In particular, a CFG in Chomsky normal form has neither λ-productions nor unit productions (which are of the form $A \to B$). Furthermore, we need a means to keep track of the values of the counters. The latter can be done by using an extra track of the input tape of the LBA for each counter, and by observing that, for a given valence grammar, there is a maximum amount m_i that can be added to (or subtracted from) the ith counter $(0 \leq i \leq k)$ during the application of a production, hence for an input word of length n the value of the ith counter becomes at most $(2n-1)m_i$ (here we again use the fact that the valence grammar is in Chomsky normal form). Thus, if necessary, the LBA can count up to $2m_i$ in each cell of the extra track corresponding to the ith counter. Hence for each $k \geq 0$, it holds

that $\mathsf{CF}(\mathbb{Z}^k) \subseteq \mathsf{CS}$.

In the same way that context-free grammars are extended to context-free valence grammars over $\mathbb{Z}^k$, for some $k \geq 0$, we can extend finite-state transducers to *valence transducers over* $\mathbb{Z}^k$ (called $\mathbb{Z}^k$-transducers for short; the mapping defined by such a transducer is called a $\mathbb{Z}^k$-transduction). Hence $\mathbb{Z}^k$-transducers are finite-state transducers in which each transition has a valence, and a computation of such a $\mathbb{Z}^k$-transducer is valid if it follows a path from the initial state to a final state and the valences along this path add to $\vec{0}$.

The application of a $\mathbb{Z}^\ell$-transduction to a valence language over $\mathbb{Z}^k$, for some $k, \ell \geq 0$, yields a valence language over $\mathbb{Z}^{k+\ell}$. This can be proved as follows (see also [FS00, Theorem 4.18]), using a construction that is similar to the 'triple construction' that defines a CFG generating the intersection of the languages of a given context-free grammar and a given finite automaton; the name 'triple construction' comes from the fact that the non-terminals of the new CFG are of the form $[p, x, q]$, where p and q are states of the automaton and x is a non-terminal or a terminal of the original CFG.

Let $G = (N, T, R, S)$ be a valence grammar over $\mathbb{Z}^k$, and let $\tau = (Q, \Sigma, \Delta, \delta, q_0, F)$ be a valence transducer over $\mathbb{Z}^\ell$. We describe now how to construct the desired valence grammar H over $\mathbb{Z}^{k+\ell}$. We may assume that G is in Chomsky normal form. The structure of a derivation tree of H is almost the same as that of the corresponding derivation tree of G, and on the way down from the root to the leaves a path through the automaton τ is guessed. This happens as follows: the start symbol S' of H makes sure that the path starts in the initial and ends in a final state, through the productions $(S' \to [q_0, S, f], \vec{0})$, for each $f \in F$. Then $(\,[p, X, q] \to [p, Y, r][r, Z, q]\,, (v_1, \ldots, v_k, 0, \ldots, 0)\,)$, where $(\,X \to YZ,\ (v_1, \ldots, v_k)\,) \in R$, recursively refines the guessed path, until $(\,[p, X, q] \to [p, a, q],\ (v_1, \ldots, v_k, 0, \ldots, 0)\,)$, for a production $(\,X \to a,\ (v_1, \ldots, v_k)\,) \in R$, terminates the derivation of G. Now $(\,[p, a, q] \to w,\ (0, \ldots, 0, r_1, \ldots, r_\ell)\,)$, for a transition (p, a, w, q) with valence $(r_1, \ldots, r_\ell)$ of τ, guarantees that indeed a path through τ has been generated and performs the transduction defined by τ.

Note that the additions to the ℓ counters of τ may happen in a different order than when the transduction defined by τ is applied directly, because this order is now dependent on the derivation of H. Since addition is commutative, this is not a problem.

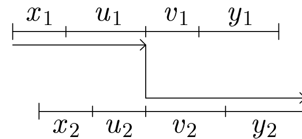
Part I

Splicing systems

Chapter 3

Definitions, examples and research topics

Analogous to the splicing of two molecules with the help of restriction enzymes to produce another molecule (see Chapter 1), two strings $x_1u_1v_1y_1$ and $x_2u_2v_2y_2$ can be spliced according to a splicing rule (u_1, v_1, u_2, v_2) to give the string $x_1u_1v_2y_2$:



Definition 3.1 A *splicing rule* over an alphabet V is an element of $(V^*)^4$. For such a rule $r = (u_1, v_1, u_2, v_2)$ and strings $x, y, z \in V^*$ we write

$$(x, y) \vdash_r z \quad \text{iff} \quad x = x_1u_1v_1y_1,\ y = x_2u_2v_2y_2 \text{ and } z = x_1u_1v_2y_2,\ \text{for some } x_1, y_1, x_2, y_2 \in V^*.$$

□

The string z is said to be obtained by splicing the strings x and y using the rule r; x is called the first term of the splicing, and y the second term.

A splicing system consists of an initial language, modelling the contents of a tube of DNA, and a set of rules, modelling the set of available restriction enzymes.

Definition 3.2 A *splicing system* (or *H system*) is a triple $h = (V, L, R)$ where V is an alphabet, $L \subseteq V^*$ is the *initial language* and $R \subseteq (V^*)^4$ is a set of splicing rules, the *splicing relation*. □

In the literature, splicing rules are usually represented as strings rather than 4-tuples: a splicing rule $r = (u_1, v_1, u_2, v_2)$ is given as the string $\mathcal{Z}(r) = u_1\#v_1\$u_2\#v_2$ ($\#$ and $\$$ are special symbols not in V), i.e., $\mathcal{Z}$ is a mapping from $(V^*)^4$ to $V^*\#V^*\$V^*\#V^*$, that gives a string representation of each splicing rule. We extend $\mathcal{Z}$ in the natural way to a mapping from sets of splicing rules to languages. The name of this mapping is suggested by a more graphical notation for the splicing rule r, that is used in [PRS96a]:

$$\begin{array}{c|c} u_1 & v_1 \\ \hline u_2 & v_2 \end{array}$$

and by the way the diagram is then read to get the $u_1\#v_1\$u_2\#v_2$ notation. These three ways to represent splicing rules are all useful: the representation by 4-tuples is 'safe' whereas the string representation may cause problems, as we explain in Chapter 4; the string representation, however, enables us to measure the complexity of a set of rules by determining its position in the Chomsky hierarchy; and the representation by diagram may be easier to read than the other two.

Let $\mathcal{F}_1$ and $\mathcal{F}_2$ be families of languages. A splicing system with $L \in \mathcal{F}_1$ and $\mathcal{Z}(R) \in \mathcal{F}_2$ is said to be of $(\mathcal{F}_1, \mathcal{F}_2)$ type. Accordingly, we will sometimes write, for instance, 'regular set of splicing rules' when we mean a set of splicing rules of which the $\mathcal{Z}$-representation is a regular language.

In natural splicing, one splicing can yield both $x_1u_1v_2y_2$ and $x_2u_2v_1y_1$. The formal language variant of this kind of splicing is called 2-splicing and is investigated in, for instance, [PRS98, Chapter 8]. Note that the splicing model we consider here can simulate 2-splicing by adding for each rule (u_1, v_1, u_2, v_2) also the symmetric rule (u_2, v_2, u_1, v_1).

3.1 Non-iterated splicing

One way to look at the splicing operation is to see it as a unary operation on languages. In other words, we can consider the language consisting of all words that are the result of splicing any two words from a given initial language using an appropriate splicing rule from a given set of splicing rules.

Definition 3.3 For a splicing system $h = (V, L, R)$

$$\sigma(h) = \{z \in V^* \mid (x, y) \vdash_r z \text{ for some } x, y \in L \text{ and } r \in R\}$$

is the *(non-iterated splicing) language* generated by h. □

As explained above, a splicing relation R is usually represented by a language $\mathcal{Z}(R)$, which gives the possibility to study the power of splicing with rules from a certain family of languages: for instance, what is the result of splicing linear initial languages with linear splicing rules? We give some initial examples.

Example 3.1 Let $h = (\{a,b,c,d\}, L, R)$ be a splicing system with

$$\begin{array}{rcl} L & = & \{a^n b^n \cdot d \mid n \geq 1\} \cup \{d \cdot b^n c^n \mid n \geq 1\} \ \in \ \mathsf{LIN} \\ Z(R) & = & \{a\#b^i d\$d\#b^i c \mid i \geq 1\} \ \in \ \mathsf{LIN} \end{array}$$

Clearly the first term of each splicing should be of the form $a^k b^k d$ and the second term of the form $d\, b^j c^j$, for some $k, j \geq 1$. Moreover, because of the form of the rules it must be that $k = j$. Thus all splicings are of the form (we indicate the cutting points with a '|')

$$(a^k \mid b^k d\ ,\ d \mid b^k c^k) \vdash a^k b^k c^k$$

Hence the language generated by h is

$$\sigma(h) = \{a^n b^n c^n \mid n \geq 1\} \ \in \ \mathsf{CS} - \mathsf{CF}.$$

□

Example 3.2 Let $h = (\{a,b,c,d\}, L, R)$ be a splicing system with

$$\begin{array}{rcl} L & = & \{a^n b^n \mid n \geq 1\} \cup \{c^n d^n \mid n \geq 1\} \ \in \ \mathsf{LIN} \\ Z(R) & = & \{\ b\#\$\#c\ \} \ \in \ \mathsf{FIN} \end{array}$$

The language generated by h is

$$\sigma(h) = \{\ a^{n_1} b^{m_1} c^{m_2} d^{n_2} \mid n_i \geq m_i \geq 1 \ (i = 1,2)\ \} \ \in \ \mathsf{CF} - \mathsf{LIN}.$$

□

Given two families of languages, $\mathcal{F}_1$ and $\mathcal{F}_2$, the family $S(\mathcal{F}_1, \mathcal{F}_2)$ of non-iterated splicing languages, obtained by splicing $\mathcal{F}_1$ languages with $\mathcal{F}_2$ rules, is defined in the obvious way:

$$S(\mathcal{F}_1, \mathcal{F}_2) = \{\sigma(h) \mid h = (V, L, R) \text{ with } L \in \mathcal{F}_1 \text{ and } Z(R) \in \mathcal{F}_2\}.$$

The families $S(\mathcal{F}_1, \mathcal{F}_2)$ are investigated in [Pău96a] and [PRS96b], for $\mathcal{F}_1$ and $\mathcal{F}_2$ in the Chomsky hierarchy. An overview of these results is presented in [HPP97]. When $S(\mathcal{F}_1, \mathcal{F}_2)$ was not found to be equal to one of the six Chomsky families, the greatest lower bound $\mathcal{F}_3$ and the smallest upper bound $\mathcal{F}_4$ among them are given: $\mathcal{F}_3 \subset S(\mathcal{F}_1, \mathcal{F}_2) \subset \mathcal{F}_4$. These results are collected in Table 1 from [HPP97], which we repeat here as Table 3.1. As an example, the optimal classification (within the Chomsky hierarchy) of splicing LIN languages with REG rules is $\mathsf{LIN} \subset S(\mathsf{LIN}, \mathsf{REG}) \subset \mathsf{CF}$.

In Chapter 5 we will show that the lower and upper bounds given here for $S(\mathsf{LIN}, \mathsf{FIN})$ and $S(\mathsf{LIN}, \mathsf{REG})$ can be replaced by the characterization $\mathsf{LIN} \oplus \mathsf{LIN}$, the family consisting of finite unions of elements from $\mathsf{LIN} \cdot \mathsf{LIN}$.

Additionally we will consider the family $S(\mathcal{F}, [1])$ of languages obtained by splicing $\mathcal{F}$ languages using rules of *radius* 1, i.e., for each splicing rule (u_1, u_2, u_3, u_4) we have $|u_i| \leq 1$ for $i = 1,2,3,4$.

<table>
<tr><th>$\mathcal{F}_1 \downarrow \ \mathcal{F}_2 \rightarrow$</th><th>FIN</th><th>REG</th><th>LIN</th><th>CF</th><th>CS</th><th>RE</th></tr>
<tr><td>FIN</td><td>FIN</td><td>FIN</td><td>FIN</td><td>FIN</td><td>FIN</td><td>FIN</td></tr>
<tr><td>REG</td><td>REG</td><td>REG</td><td>REG, LIN</td><td>REG, CF</td><td>REG, RE</td><td>REG, RE</td></tr>
<tr><td>LIN</td><td>LIN, CF</td><td>LIN, CF</td><td colspan="4" rowspan="4">RE</td></tr>
<tr><td>CF</td><td>CF</td><td>CF</td></tr>
<tr><td>CS</td><td colspan="2" rowspan="2"></td></tr>
<tr><td>RE</td></tr>
</table>

Table 3.1: The position of $S(\mathcal{F}_1, \mathcal{F}_2)$ in the Chomsky hierarchy

3.2 Iterated splicing

Above we have considered splicing as an operation on languages. Another way to view the splicing operation is as a language generating mechanism: starting from a given initial language L and a given set of splicing rules R, the resulting language is the smallest language that contains L and that is closed under splicing with rules from R.

Definition 3.4 The *(iterated splicing) language* $\sigma^*(h)$ generated by a splicing system $h = (V, L, R)$ is defined by

$$\begin{aligned} \sigma^0(h) &= L \\ \sigma^{i+1}(h) &= \sigma^i(h) \ \cup \ \sigma(\sigma^i(h)), \ i \geq 0 \\ \sigma^*(h) &= \bigcup_{i \geq 0} \sigma^i(h) \end{aligned}$$

□

Note that in general $\sigma^1(L) = L \cup \sigma(L) \neq \sigma(L)$.

Similar to the non-iterated case, families of iterated splicing languages are defined by

$$H(\mathcal{F}_1, \mathcal{F}_2) = \{\sigma^*(h) \mid h = (V, L, R) \text{ with } L \in \mathcal{F}_1 \text{ and } \mathcal{Z}(R) \in \mathcal{F}_2\}$$

for $\mathcal{F}_1, \mathcal{F}_2 \in \{\mathsf{FIN}, \mathsf{REG}, \mathsf{LIN}, \mathsf{CF}, \mathsf{CS}, \mathsf{RE}\}$. A first notable result was obtained in [CH91], namely that iterated splicing of REG languages by FIN rules does not lead outside REG: $H(\mathsf{REG}, \mathsf{FIN}) = \mathsf{REG}$. The families $H(\mathcal{F}_1, \mathcal{F}_2)$ were further investigated in [Pix96], [Pău96a], [Pix95] and [Pău96b], and the results are listed in Table 2 from [HPP97], which we repeat here as Table 3.2.

Inspecting Table 3.2, we see that, e.g., splicing systems of $(\mathsf{REG}, \mathsf{LIN})$ type generate all regular languages but not all linear languages, and at least one non-context-sensitive language. In the following example we give a splicing system of $(\mathsf{REG}, \mathsf{LIN})$ type that generates a non-context-free context-sensitive language.

$\mathcal{F}_1 \downarrow\ \mathcal{F}_2 \rightarrow$	FIN	REG	LIN	CF	CS	RE
FIN	FIN, REG	FIN, RE	FIN, RE	FIN, RE	FIN, RE	FIN, RE
REG	REG	REG, RE	REG, RE	REG, RE	REG, RE	REG, RE
LIN	LIN, CF	LIN, RE	LIN, RE	LIN, RE	LIN, RE	LIN, RE
CF	CF	CF, RE	CF, RE	CF, RE	CF, RE	CF, RE
CS	CS, RE	CS, RE	CS, RE	CS, RE	CS, RE	CS, RE
RE	RE	RE	RE	RE	RE	RE

Table 3.2: The position of $H(\mathcal{F}_1, \mathcal{F}_2)$ in the Chomsky hierarchy

Example 3.3 Consider the splicing system $h = (\{a, b\}, L, R)$ defined by

$$\begin{array}{rcl} L & = & ba^+b \ \in \ \mathsf{REG} \\ Z(R) & = & \{ba^nb\,\#\,\$\,b\,\#\,a^nb \mid n \geq 1\} \ \in \ \mathsf{LIN} \end{array}$$

Then it is easily seen that $\sigma^0(h) = ba^+b$, $\sigma^1(h) = \{ba^nb, ba^nba^nb \mid n \geq 1\}$, $\sigma^2(h) = \{(ba^n)^jb \mid 1 \leq j \leq 4, n \geq 1\}$ and in general $\sigma^i(h) = \{(ba^n)^jb \mid 1 \leq j \leq 2^i, n \geq 1\}$ for each $i \geq 0$. Consequently

$$\sigma^*(h) = \{(ba^n)^kb \mid n, k \geq 1\},$$

which is clearly not context-free. In fact, $\sigma^*(h) \in \mathsf{CS} - \mathsf{CF}$, since an LBA can compare each pair of two adjacent groups of a's (separated by a b) and in this way check that all groups of a's are of the same length. □

An interesting problem suggested by the information from Table 3.2 is to find an algorithm to determine whether a regular language is in $H(\mathsf{FIN}, \mathsf{FIN})$. Our attempt in this direction led to the following examples and to Theorem 3.1, that states that each regular language can be generated by a finite splicing system provided that we let every string be preceded by a special marker. This was already observed in [Hea98, remark following Theorem 3.1], but we repeat it here, and give a different proof.

Example 3.4 Let $h = (\{a, b\}, L, R)$ be the splicing system defined by

$$\begin{array}{rcl} L & = & \{\, \lambda\, ,\ a\, ,\ b\, ,\ aa\, ,\ ba \,\} \ \in \ \mathsf{FIN} \\ R & = & \left\{ \begin{array}{c|c} b & \\ \hline & ab \end{array}\, ,\ \begin{array}{c|c} b & \\ \hline & b \end{array}\, ,\ \begin{array}{c|c} a & \\ \hline & b \end{array}\, ,\ \begin{array}{c|c} aa & \\ \hline & a \end{array} \right\} \ \in \ \mathsf{FIN} \end{array}$$

Then $\sigma^*(h) = \{w \in \{a, b\}^* \mid w$ does not contain $baa\}$, which is a regular language.

The idea behind this is that we concatenate two (non-empty) words of which we already know that they do not contain baa. Then we only have to ensure

that such a concatenation does not create an occurrence of baa. Suppose that the first word ends with a b. If the second word starts with an a, then we have to be sure that directly after this a there is not another a – this gives the rule $(b, \lambda, \lambda, ab)$. If the second word starts with a b, then there is no problem – we add the rule (b, λ, λ, b). Now suppose that the first word ends with an a. For the case that the second word starts with a b, we add the rule (a, λ, λ, b). However, if the second word starts with an a, then we must guarantee that the first word does not end with ba – this gives the rule $(aa, \lambda, \lambda, a)$.

With these rules we cannot generate the words λ, a, b, aa and ba, so they form the initial language. Then we have

$$(a \mid , \mid b) \vdash_{(a,\lambda,\lambda,b)} ab \qquad \text{and} \qquad (b \mid , \mid b) \vdash_{(b,\lambda,\lambda,b)} bb,$$

thus $\sigma^*(h)$ contains all words of length 2 or smaller, which should indeed be the case. Since each word that does not contain baa can arbitrarily be decomposed into two subwords that do not contain baa, and since the rules are constructed such that they represent all possibilities to concatenate two such subwords, now all words not containing baa can be generated recursively from shorter words.

□

Example 3.5 The regular language $a^*ba^*ba^*$ is not in $H(\mathsf{FIN}, \mathsf{FIN})$: splicing systems cannot distinguish between the two b's, because there are arbitrarily large numbers of a's before, between and after them. More specifically, if one of the two splicing sites of a rule contains less than two b's, then the resulting word may contain more or less than two b's, whereas if each splicing site has to contain exactly two b's, then the number of a's in between is bounded.

However, a^*ba^*b is in $H(\mathsf{FIN}, \mathsf{FIN})$, since $\sigma^*(h) = a^*ba^*b$ for the finite splicing system $h = (\{a, b\}, L, R)$ with

$$\begin{aligned} L &= \{bb\,,\ abb\,,\ bab\,,\ abab\} \\ R &= \left\{ \frac{a \mid bb}{\ \mid abb}\,,\ \frac{a \mid ba}{\ \mid aba}\,,\ \frac{ba \mid\ }{b \mid a} \right\} \end{aligned}$$

The reason for this is that in this case the two b's *can* be told apart: the first is followed by an a or a b, the second is not. Therefore, the strings ba and bb can be used as two distinct 'handles' or 'fixed points' from where the arbitrarily large numbers of a's can be generated. (See also [Hea98, p.276].) □

The idea behind the following theorem, that states that any regular language preceded by a new symbol is in $H(\mathsf{FIN}, \mathsf{FIN})$, is that this new symbol can be used as a handle. We explain this through an example.

Example 3.6 Consider the language $ca^*ba^*ba^* \in \mathsf{REG}$. This language can be generated by a finite splicing system $h = (\{a,b,c\}, L, R)$ defined as follows:

$$L = \{\, ca^iba^jba^k \mid i,j,k \in \{0,1\}\,\}$$
$$R = \left\{ \frac{ca^iba^jba \,|}{ca^iba^jb \,|}, \frac{ca^iba \,|}{ca^iba^jbca^ib \,|}, \frac{ca \,|}{c \,|} \;\mid\; i,j \in \{0,1\} \right\}$$

Now each of the four rules $(ca^iba^jba, \lambda, ca^iba^jb, \lambda)$, for $i,j \in \{0,1\}$, can be used to generate the arbitrarily many a's in the third group:

$$(ca^iba^jba \mid , \; ca^iba^jb \mid a^n) \vdash ca^iba^jba^{n+1} \quad \text{for each } n \geq 0.$$

Similarly, the a's in the second and first group (in this order) are generated using the rules $(ca^iba, \lambda, ca^ib, \lambda)$, with $i \in \{0,1\}$, and $(ca, \lambda, c, \lambda)$, respectively:

$$(ca^iba \mid b \,, \; ca^ib \mid a^nba^m) \vdash ca^iba^{n+1}ba^m \quad \text{for each } n,m \geq 0,$$

and

$$(ca \mid bb \,, \; c \mid a^nba^mba^k) \vdash ca^{n+1}ba^mba^k \quad \text{for each } n,m,k \geq 0.$$

□

We need some definitions related to classical 'pumping properties' of finite automata. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton. We call $q_0q_1 \ldots q_m$ a *state sequence* (for $a_0a_1 \ldots a_{m-1}$) in $\mathcal{A}$ if $q_j \in Q$ for $0 \leq j \leq m$, $m \geq 0$, $q_m \in F$ and there are $a_i \in \Sigma$ such that $\delta(q_i, a_i) = q_{i+1}$, for $0 \leq i < m$. A state sequence $q_0 \ldots q_i \ldots q_j \ldots q_m$ in $\mathcal{A}$ with $0 \leq i < j \leq m$ *reduces* to a state sequence $q_0 \ldots q_iq_{j+1} \ldots q_m$ in $\mathcal{A}$ if $q_i = q_j$ and j is the smallest index such that there is an $\ell < j$ with $q_\ell = q_j$.

A word $w \in L(\mathcal{A})$ *reduces* to a word $w' \in L(\mathcal{A})$, denoted $w \succ w'$, if the state sequence for w in $\mathcal{A}$ reduces to the state sequence for w' in $\mathcal{A}$.

Theorem 3.1 *Let K be a regular language over Σ, and let c be a symbol not in Σ. Then $cK \in H(\mathsf{FIN}, \mathsf{FIN})$.*

Proof. *Construction.* Since REG is closed under concatenation with symbols, there is a DFA $\mathcal{A} = (Q, \Sigma \cup \{c\}, \delta, q_0, F)$ with $L(\mathcal{A}) = cK$. We construct a finite splicing system $h = (\Sigma \cup \{c\}, L, R)$ such that $\sigma^*(h) = L(\mathcal{A})$ as follows.

$$L = \{\, w \in L(\mathcal{A}) \mid \text{the state sequence for } w \text{ in } \mathcal{A} \text{ does not contain three occurrences of the same state}\,\}$$
$$R = \left\{ \frac{cuv \,|}{cu \,|} \;\mid\; cuvw \in L \text{ for a } w \in \Sigma^*,\ v \neq \lambda,\ \delta(q_0, cu) = q = \delta(q, v) \text{ for a } q \in Q \,\right\}$$

It is clear that L and R are finite.

Correctness. A proof that $\sigma^*(h) \subseteq L(\mathcal{A})$ uses induction on σ^i. For $i = 0$ we have $\sigma^0(h) = L \subseteq L(\mathcal{A})$ by the definition of L. Now assume that $\sigma^i(h) \subseteq L(\mathcal{A})$ for a certain $i \geq 0$, and observe the language $\sigma^{i+1}(h) = \sigma^i(h) \cup \sigma(\sigma^i(h))$. Suppose that x and y in $\sigma^i(h)$ are spliced using the rule $(cuv, \lambda, cu, \lambda)$ from R, hence $x = cuvw$ and $y = cuz$ for some $w, z \in \Sigma^*$. From the definition of R we know that $\delta(q_0, cu) = q$ and $\delta(q, v) = q$, for some $q \in Q$. Since y is in $\sigma^i(h)$ and thus in $L(\mathcal{A})$, and since $\mathcal{A}$ is deterministic, we also have $\delta(q, z) \in F$. Thus $cuvz$, which is the result of splicing $x = cuvw$ and $y = cuz$ using $(cuv, \lambda, cu, \lambda)$, is in $L(\mathcal{A})$. Consequently $\sigma^{i+1}(h) \subseteq L(\mathcal{A})$, hence $\sigma^*(h) \subseteq L(\mathcal{A})$.

For a proof of $L(\mathcal{A}) \subseteq \sigma^*(h)$ first note that for every word $z_0 \in L(\mathcal{A})$ a reduction $z_0 \succ z_1 \succ \ldots \succ z_k$, where $k \geq 0$, exists such that the state sequence for z_k does not contain any state twice.

We claim that this reduction can be carried out in the reverse order by h, which means that any word in $L(\mathcal{A})$ can be created starting from words in L and using splicing rules from R. Obviously, $z_k \in L$ and thus $z_k \in \sigma^*(h)$. Now assume that $z_\ell \in \sigma^*(h)$ for some ℓ with $0 < \ell \leq k$, and consider $z_{\ell-1}$. From the reduction step $z_{\ell-1} \succ z_\ell$ we know that there are $u, v, w \in \Sigma^*$ such that $z_{\ell-1} = cuvw$, $v \neq \lambda$, $\delta(q_0, cu) = q$, $\delta(q, v) = q$, $\delta(q, w) \in F$ and the second occurrence of q is the first repetition in the state sequence for $z_{\ell-1}$ in $\mathcal{A}$. Then $z_\ell = cuw$. Furthermore, since it is possible to reach a final state from q, this can also be done without passing any state twice. Hence there is a $w' \in \Sigma^*$ such that $cuvw'$ is in L and thus $r = (cuv, \lambda, cu, \lambda)$ is in R. Now $(cuv \mid w' \,,\, cu \mid w) \vdash_r cuvw = z_{\ell-1}$, and since $cuvw'$ is in L and we assumed that $cuw = z_\ell$ is in $\sigma^*(h)$ we have $z_{\ell-1} \in \sigma^*(h)$. □

Note the similarities in the above proof with the pumping lemma for regular languages, that states that if K is a regular language, then there is an $n \geq 1$ such that for each $z \in K$ with $|z| \geq n$ there exist u, v, w such that $z = uvw$, $v \neq \lambda$, $|uv| \leq n$ and $uv^iw \in K$ for all $i \geq 0$. In the part of the proof where we show that $\sigma^*(h) \subseteq L(\mathcal{A})$, the splicing of $x = cuvw$ and $y = cuz$ using $(cuv, \lambda, cu, \lambda)$ to give $cuvz$ can be seen as the extension of cuz to $cuvz$, after which the splicing of two occurrences of $cuvz$ using the same rule yields cuv^2z, and so on (i.e., we 'pump up'). On the other hand, for the reduction step $cuvw = z_{\ell-1} \succ z_\ell = cuw$ used in the part of the proof that demonstrates that $L(\mathcal{A}) \subseteq \sigma^*(h)$ it holds that $cuw = cuv^0w$ (i.e., we 'pump down').

A solution to the problem of deciding whether a regular language is in $H(\mathsf{FIN}, \mathsf{FIN})$ can be found in [BZ99].

3.3 Restricted non-iterated splicing

In this section we consider the setting where the general splicing operation $(x, y) \vdash_r z$ may only be applied in a certain context.

We start by recalling the definitions of certain types of restricted splicing from [PRS96b, KPS96]. We splice in *length-increasing* mode (*in* for short) if the length of the resulting string is strictly greater than the lengths of the two input strings, in *length-decreasing* mode (*de*) if the length of the resulting string is strictly smaller than the lengths of the two input strings, in *same-length* mode (*sl*) if the two input strings have the same length, and in *self splicing* mode (*sf*) if the two input strings are equal. Formally, for a splicing rule r

$$\begin{aligned}
(x,y) \vdash_r^{in} z &\quad \text{iff} \quad (x,y) \vdash_r z \text{ and } |z| > \max\{|x|,|y|\} \\
(x,y) \vdash_r^{de} z &\quad \text{iff} \quad (x,y) \vdash_r z \text{ and } |z| < \min\{|x|,|y|\} \\
(x,y) \vdash_r^{sl} z &\quad \text{iff} \quad (x,y) \vdash_r z \text{ and } |x| = |y| \\
(x,y) \vdash_r^{sf} z &\quad \text{iff} \quad (x,y) \vdash_r z \text{ and } x = y.
\end{aligned}$$

Let $h = (V, L, R)$ be a splicing system. With the restricted splicing operations given above we define the (restricted non-iterated splicing) languages

$$\sigma_\mu(h) = \{z \in V^* \mid (x,y) \vdash_r^\mu z \text{ for some } x,y \in L \text{ and } r \in R\}$$

for $\mu \in \{in, de, sl, sf\}$. Similarly we define the families

$$S_\mu(\mathcal{F}_1, \mathcal{F}_2) = \{\sigma_\mu(h) \mid h = (V, L, R) \text{ with } L \in \mathcal{F}_1 \text{ and } \angle(R) \in \mathcal{F}_2\}.$$

We mainly consider $\mathcal{F}_1 = \mathsf{REG}, \mathsf{LIN}, \mathsf{CF}$ and $\mathcal{F}_2 = \mathsf{FIN}, \mathsf{REG}, \mathsf{LIN}, \mathsf{CF}$. We repeat in Table 3.3 the parts of Tables 1, 2 and 3 from [KPS96] that give the lowest upper bounds within the Chomsky hierarchy for the families $S_\mu(\mathcal{F}_1, \mathcal{F}_2)$ for the modes μ that we consider. For comparison we repeat in the first row of the table the smallest upper bounds for unrestricted non-iterated splicing (also called *free* splicing, *f* for short) given in Table 3.1.

$\mathcal{F}_2 \rightarrow$	FIN	REG	LIN	CF	FIN	REG	LIN	CF
f	REG	REG	LIN	CF	CF	CF	RE	RE
in	REG	REG		CF$^+$	CS	CS	CS	CS
de	REG	REG		CF$^+$	CS	CS	RE	RE
sl	LIN	LIN	CF$^+$		CF$^+$		RE	RE
sf	CS	CS			CF$^+$		RE	RE
	$\mathcal{F}_1 = \mathsf{REG}$				$\mathcal{F}_1 = \mathsf{LIN}, \mathsf{CF}$			

Table 3.3: Smallest upper bounds of $S_\mu(\mathcal{F}_1, \mathcal{F}_2)$ within the Chomsky hierarchy

For the families corresponding to the entries marked with CF^+ it is only known that the family contains a non-context-free language; it is not yet determined whether the smallest upper bound within the Chomsky hierarchy is CS or RE.

Note that although, for instance, the table contains the same values for the families $S_{sl}(\mathsf{REG}, \mathsf{FIN})$ and $S_{sl}(\mathsf{REG}, \mathsf{REG})$, this does not necessarily mean

that they are equal: they only have the same upper bound in the Chomsky hierarchy. The same remark holds for the equality of the tables for $\mathcal{F}_1 = \mathsf{LIN}$ and $\mathcal{F}_1 = \mathsf{CF}$.

Examples of restricted splicing in these four modes are given in Chapters 5 and 6.

3.4 Research topics

In the literature on splicing systems, a splicing rule (u_1, v_1, u_2, v_2) is usually represented by the string $u_1\#v_1\$u_2\#v_2$, and so a set of splicing rules is a language. Although this string representation of splicing rules is very natural, other representations are possible. The question arises whether the results on the generative power of splicing with rules from a certain family of languages, that are mentioned in the literature, are properties of the *splicing systems* or of the specific *representation* of splicing rules that is chosen. For example, do we get different results if we first write the left contexts of the rule and then the right contexts, choosing $u_1\#u_2\$v_1\#v_2$ instead of $u_1\#v_1\$u_2\#v_2$ as the string representation of (u_1, v_1, u_2, v_2) ? In Chapter 4 we answer this question in detail for non-iterated and iterated splicing systems, and show that the classifications in Tables 3.1 and 3.2 are not influenced by this particular change in representation. We briefly discuss some other, related, string representations.

The first two columns of Table 3.1 show that $S(\mathcal{F}, \mathsf{FIN}) = S(\mathcal{F}, \mathsf{REG})$ for all families $\mathcal{F}$ considered here except for LIN, for which it is only known that $S(\mathsf{LIN}, \mathsf{FIN})$ and $S(\mathsf{LIN}, \mathsf{REG})$ have the same upper and lower bounds within the Chomsky hierarchy. In Chapter 5 we show for all $\mathcal{F}$ except CS how the regular rule set may be replaced by a finite one, i.e., we give direct proofs of the equalities $S(\mathcal{F}, \mathsf{REG}) = S(\mathcal{F}, \mathsf{FIN})$. Moreover, we show that $S(\mathcal{F}, \mathsf{FIN}) = \mathcal{F} \oplus \mathcal{F}$ – i.e., it consists of all finite unions of concatenations of two languages in $\mathcal{F}$– and thus obtain the new result that both $S(\mathsf{LIN}, \mathsf{FIN})$ and $S(\mathsf{LIN}, \mathsf{REG})$ are characterized by the family $\mathsf{LIN} \oplus \mathsf{LIN}$. Furthermore, we try to extend the latter result to the case of restricted non-iterated splicing.

For several modes μ of restricted splicing and for certain families of languages $\mathcal{F}_1$ and $\mathcal{F}_2$, it is not yet known what the smallest upper bound within the Chomsky hierarchy is for splicing $\mathcal{F}_1$ languages in mode μ using rule sets from $\mathcal{F}_2$ (see Table 3.3). In Chapter 6 we solve the open problems in this table, and moreover we improve some of the upper bounds given there from CS to $\mathsf{CF}(\mathbb{Z}^k)$, the family of context-free valence languages over $\mathbb{Z}^k$, for either $k = 1$ or $k = 2$.

Chapter 4

String representations of splicing rules

In most of the literature on splicing systems a splicing rule $r = (u_1, v_1, u_2, v_2)$ is represented by the string $\mathsf{Z}(r) = u_1\#v_1\$u_2\#v_2$. In this way a set of splicing rules becomes a language and consequently its complexity can be measured by determining its position in the Chomsky hierarchy. We investigate whether taking a string representation of splicing rules other than the standard one has any effect on the position in the Chomsky hierarchy of the families of non-iterated and iterated splicing languages.

To allow for other string representations, we extend the notation for families of splicing languages $S(\mathcal{F}_1, \mathcal{F}_2)$ and $H(\mathcal{F}_1, \mathcal{F}_2)$. Let $\rho : (V^*)^4 \to W^*$ be a given string representation of splicing rules over the alphabet V, for some alphabet W. Then define

$$S_\rho(\mathcal{F}_1, \mathcal{F}_2) = \{\sigma(h) \mid h = (V, L, R), \text{ with } L \in \mathcal{F}_1 \text{ and } \rho(R) \in \mathcal{F}_2\}$$

and

$$H_\rho(\mathcal{F}_1, \mathcal{F}_2) = \{\sigma^*(h) \mid h = (V, L, R), \text{ with } L \in \mathcal{F}_1 \text{ and } \rho(R) \in \mathcal{F}_2\}.$$

Hence, by definition, $S(\mathcal{F}_1, \mathcal{F}_2) = S_{\mathsf{Z}}(\mathcal{F}_1, \mathcal{F}_2)$ and $H(\mathcal{F}_1, \mathcal{F}_2) = H_{\mathsf{Z}}(\mathcal{F}_1, \mathcal{F}_2)$ for the standard string representation Z.

We will also directly consider the family of *splicing relations* defined by the family $\mathcal{F}$ of languages over W under the representation $\rho : (V^*)^4 \to W^*$,

$$\mathcal{R}_\rho(\mathcal{F}) = \{R \subseteq (V^*)^4 \mid \rho(R) \in \mathcal{F}\}.$$

Now consider an alternative representation: first writing the left contexts, and then the right contexts of the splicing sites. Formally we use the mapping $\text{И} : (V^*)^4 \to V^*\#V^*\$V^*\#V^*$ defined by $\text{И}(u_1, v_1, u_2, v_2) = u_1\#u_2\$v_1\#v_2$.

In Section 4.1 we investigate whether the splicing relations defined by the language families from the Chomsky hierarchy are changed when we move from

the Z-representation to the И-representation. In other words, for each $\mathcal{F}_2$ in the Chomsky hierarchy, we determine whether or not $\mathcal{R}_{\mathsf{Z}}(\mathcal{F}_2) = \mathcal{R}_{\text{И}}(\mathcal{F}_2)$.

It turns out that for $\mathcal{F}_2 \in \{\mathsf{FIN}, \mathsf{REG}, \mathsf{CS}, \mathsf{RE}\}$ indeed $\mathcal{R}_{\mathsf{Z}}(\mathcal{F}_2) = \mathcal{R}_{\text{И}}(\mathcal{F}_2)$. Obviously this implies for all $\mathcal{F}_1$ that $S_{\mathsf{Z}}(\mathcal{F}_1, \mathcal{F}_2) = S_{\text{И}}(\mathcal{F}_1, \mathcal{F}_2)$ and $H_{\mathsf{Z}}(\mathcal{F}_1, \mathcal{F}_2) = H_{\text{И}}(\mathcal{F}_1, \mathcal{F}_2)$.

For $\mathcal{F}_2 \in \{\mathsf{LIN}, \mathsf{CF}\}$, for which we show in Subsection 4.1.1 that $\mathcal{R}_{\mathsf{Z}}(\mathcal{F}_2) \neq \mathcal{R}_{\text{И}}(\mathcal{F}_2)$, we prove that nevertheless $S_{\mathsf{Z}}(\mathcal{F}_1, \mathcal{F}_2) = S_{\text{И}}(\mathcal{F}_1, \mathcal{F}_2)$ for all $\mathcal{F}_1$ in the Chomsky hierarchy (Section 4.2), while in the case of iterated splicing we only show that the smallest upper bounds and greatest lower bounds given for $H_{\mathsf{Z}}(\mathcal{F}_1, \mathcal{F}_2)$ in Table 3.2 also hold for $H_{\text{И}}(\mathcal{F}_1, \mathcal{F}_2)$ (Section 4.3). The precise relation between $H_{\mathsf{Z}}(\mathcal{F}_1, \mathcal{F}_2)$ and $H_{\text{И}}(\mathcal{F}_1, \mathcal{F}_2)$ in these cases is open for further investigation.

In Section 4.4 we discuss related string representations.

4.1 Families of splicing relations

In this section we compare the families of splicing relations defined by the two string representations Z and И of splicing rules.

Observe that representations like Z and И define a one-to-one correspondence between a splicing rule (u, v, w, x) and its string representations $u\#v\$w\#x$ and $u\#w\$v\#x$, respectively. Thus when considering such a representation ρ of a splicing relation R over the alphabet V and a language $L \subseteq V^*\#V^*\$V^*\#V^*$, one has $\rho(R) = L$ if and only if $R = \rho^{-1}(L)$. Consequently we may write

$$\mathcal{R}_\rho(\mathcal{F}) = \{\rho^{-1}(L) \mid L \subseteq V^*\#V^*\$V^*\#V^* \text{ and } L \in \mathcal{F}\}.$$

Hence proving that $\mathcal{R}_{\mathsf{Z}}(\mathcal{F}) \subseteq \mathcal{R}_{\text{И}}(\mathcal{F})$ amounts to verifying that $R = \mathsf{Z}^{-1}(L)$ for an $L \in \mathcal{F}$ implies $\text{И}(R) = K$ for a $K \in \mathcal{F}$. Consequently we have to prove that $\text{И}(\mathsf{Z}^{-1}(L)) \in \mathcal{F}$ for every $L \in \mathcal{F}$ with $L \subseteq V^*\#V^*\$V^*\#V^*$. Note that this is a closure property of the family $\mathcal{F}$, namely closure under the operation $\text{И}\mathsf{Z}^{-1}$ that maps a string $u\#v\$w\#x$ to the string $u\#w\$v\#x$.

Also note that this operation is its own inverse, which implies that $\text{И}\mathsf{Z}^{-1} = \mathsf{Z}\text{И}^{-1}$ since $\text{И}\mathsf{Z}^{-1} = (\text{И}\mathsf{Z}^{-1})^{-1} = (\mathsf{Z}^{-1})^{-1}\text{И}^{-1} = \mathsf{Z}\text{И}^{-1}$. This means that $\mathcal{R}_{\mathsf{Z}}(\mathcal{F}) \subseteq \mathcal{R}_{\text{И}}(\mathcal{F})$ implies the converse inclusion $\mathcal{R}_{\text{И}}(\mathcal{F}) \subseteq \mathcal{R}_{\mathsf{Z}}(\mathcal{F})$.

4.1.1 LIN and CF splicing rules

It is easy to see that $\mathcal{R}_{\mathsf{Z}}(\mathsf{CF}) \neq \mathcal{R}_{\text{И}}(\mathsf{CF})$: the set of rules $R = \{(a^n, b^m, a^n, b^m) \mid m, n \geq 1\}$ is not context-free when the classical Z-representation is used. If we use the new И-representation instead, then $\text{И}(R) = \{\, a^n \# a^n \$ b^m \# b^m \mid m, n \geq 1\}$ is a context-free language.

We now show the same inequality for LIN. Consider the splicing relation $R = \{(a^p, c^q, b^r, d^s) \mid p, q, r, s \geq 1 \text{ and } p+q = r+s\}$. Then the Z-representation of R is a linear language, as demonstrated in the following example.

Example 4.1 Let G be the linear CFG with start symbol S defined by

$$\begin{aligned} S &\to aSd \mid a\#Td \mid aU\#d \\ T &\to cTd \mid cVb\# \\ U &\to aUb \mid \#cVb \\ V &\to cVb \mid \$ \end{aligned}$$

It can easily be verified that $L(G) = \text{Z}(R) = \{a^p\#c^q\$b^r\#d^s \mid p,q,r,s \geq 1$ and $p+q = r+s\}$. □

The И-representation of R, however, is not linear. To prove that, we use Lemma 2 from [Gre79], which we repeat here.

Proposition 4.1 *Let $L \subseteq a^+b^+c^+d^+$ be a language such that*

1. *$a^nb^nc^kd^k \in L$ for all $n,k \geq 1$,*
2. *if $a^nb^nc^kd^\ell$ is in L, then $k \leq \ell$, and*
3. *there are integers $t_1, t_2 \geq 1$ such that, if $a^nb^mc^kd^\ell$ is in L and $n > m$, then $(n-m)t_1 \leq (k+\ell)t_2$.*

Then L is not linear context-free.

Lemma 4.1 *$K = \{a^pb^rc^qd^s \mid p,q,r,s \geq 1$ and $p+q = r+s\} \notin \mathsf{LIN}$.*

Proof. Proposition 4.1 is applicable, because obviously $K \subseteq a^+b^+c^+d^+$; (1) if $p = r$ and $q = s$, then $p+q = r+s$; (2) if $p = r$ and we must have $p+q = r+s$, then it has to be the case that $q = s$; (3) taking $t_1 = t_2 = 1$, we see that if $p > r$ and $p+q = r+s$, then $0 < p-r = -q+s$, hence $p-r \leq q+s$. Consequently, K is not linear context-free. □

Since LIN is closed under homomorphisms, from Lemma 4.1 it follows that $\text{И}(R) = \{a^p\#b^r\$c^q\#d^s \mid p,q,r,s \geq 1$ and $p+q = r+s\} \notin \mathsf{LIN}$ and consequently $\mathcal{R}_{\text{Z}}(\mathsf{LIN}) \neq \mathcal{R}_{\text{И}}(\mathsf{LIN})$.

Theorem 4.2 *$\mathcal{R}_{\text{Z}}(\mathcal{F}) \neq \mathcal{R}_{\text{И}}(\mathcal{F})$ for $\mathcal{F} = \mathsf{LIN}, \mathsf{CF}$.*

Consequently, for LIN and CF splicing rules, it matters which string representation is used.

4.1.2 FIN, REG, CS and RE splicing rules

For the finite languages, it should be clear that the two representations are equivalent: if R is a finite splicing relation, then both $\text{Z}(R)$ and $\text{И}(R)$ are finite languages.

For the regular, context-sensitive and recursively enumerable languages we use the following lemma.

Lemma 4.3 *$\mathsf{REG}, \mathsf{CS}$ and RE are closed under $\text{И}\,\text{Z}^{-1}$.*

Proof. The mapping $\text{И}\,\text{Z}^{-1}$ can be realized by a 2-way deterministic generalised sequential machine (2DGSM, a finite-state device with a 2-way input tape and a 1-way output tape, see [AU70]): on input $u\#v\$w\#x$ it outputs $u\#$, skips $v\$$, outputs $w\$$, returns on the input to the first $\#$, outputs $v\#$, skips $\$w\#$ and finally outputs x. Since $\text{И}\,\text{Z}^{-1}$ is its own inverse, it can also be realized by an inverse 2DGSM mapping. The result now follows from the closure of REG, CS and RE under inverse 2DGSM mappings, see [AU70, Theorem 2]. □

As observed before, this closure property implies the following equalities.

Lemma 4.4 *$\mathcal{R}_{\text{Z}}(\mathcal{F}) = \mathcal{R}_{\text{И}}(\mathcal{F})$ for $\mathcal{F} = \mathsf{FIN}, \mathsf{REG}, \mathsf{CS}, \mathsf{RE}$.*

Consequently for splicing systems with $\mathsf{FIN}, \mathsf{REG}, \mathsf{CS}$ or RE splicing rules it does *not* matter whether we use Z or И as string representation.

4.2 Families of non-iterated splicing languages

We know that $\mathcal{R}_{\text{Z}}(\mathcal{F}_2) = \mathcal{R}_{\text{И}}(\mathcal{F}_2)$ for $\mathcal{F}_2 \in \{\mathsf{FIN}, \mathsf{REG}, \mathsf{CS}, \mathsf{RE}\}$, and thus that for each of the six families $\mathcal{F}_1$ considered here the following holds.

Theorem 4.5 *$S_{\text{Z}}(\mathcal{F}_1, \mathcal{F}_2) = S_{\text{И}}(\mathcal{F}_1, \mathcal{F}_2)$ for $\mathcal{F}_2 = \mathsf{FIN}, \mathsf{REG}, \mathsf{CS}, \mathsf{RE}$.*

For $\mathcal{F}_2 \in \{\mathsf{LIN}, \mathsf{CF}\}$, however, we have demonstrated that $\mathcal{R}_{\text{Z}}(\mathcal{F}_2) \neq \mathcal{R}_{\text{И}}(\mathcal{F}_2)$, and consequently, we still have to investigate the situation for these two possibilities for $\mathcal{F}_2$. Because in Table 3.1 exact classifications are given for $S(\mathcal{F}, \mathsf{LIN})$ and $S(\mathcal{F}, \mathsf{CF})$ for each $\mathcal{F}$ except for REG, we consider the splicing of non-REG languages apart from the splicing of REG languages.

Theorem 4.6 *$S_{\text{Z}}(\mathcal{F}_1, \mathcal{F}_2) = S_{\text{И}}(\mathcal{F}_1, \mathcal{F}_2)$ for $\mathcal{F}_1 \neq \mathsf{REG}$ and $\mathcal{F}_2 = \mathsf{LIN}, \mathsf{CF}$.*

Proof. We show that the results used in [HPP97] to determine the position of $S_{\text{Z}}(\mathcal{F}_1, \mathcal{F}_2)$ with $\mathcal{F}_1 \neq \mathsf{REG}$ and $\mathcal{F}_2 \in \{\mathsf{LIN}, \mathsf{CF}\}$ in the Chomsky hierarchy also hold when the И-representation is used. These results are the following:

(1) $S_{\text{Z}}(\mathsf{FIN}, \mathcal{F}_2) \subseteq \mathsf{FIN}$ (obvious),

(2) $\mathcal{F}_1 \subseteq S_{\text{Z}}(\mathcal{F}_1, \mathcal{F}_2)$ [HPP97, Lemma 3.2],

(3) $L_1/L_2 \in S_{\text{Z}}(\mathcal{F}_2, \mathcal{F}_2)$ for each $L_1, L_2 \in \mathcal{F}_2$ [HPP97, Lemma 3.7].

The proof of (1) is independent of the splicing rules, therefore $S_{\text{И}}(\mathsf{FIN}, \mathcal{F}_2) \subseteq \mathsf{FIN}$ holds. In the proof of (2), only one splicing rule is used, (λ, c, c, λ), for which the Z-representation is equal to the И-representation. For the splicing relation used to prove (3), which is $R = \{(\lambda, wc, c, \lambda) \mid w \in L_2\}$, where $L_2 \in \mathcal{F}_2, L_2 \subseteq V^*$

and $c \notin V$, it should be clear that both $\mathsf{Z}(R) = \#L_2c\$c\#$ and $\text{И}(R) = \#c\$L_2c\#$ belong to $\mathcal{F}_2$.

By (1) and (2) we have $\mathsf{FIN} \subseteq S_{\text{И}}(\mathsf{FIN}, \mathcal{F}_2) \subseteq \mathsf{FIN}$. As each RE-language is the quotient of two linear languages ([LLR85]), from (3) the inclusion $\mathsf{RE} \subseteq S_{\text{И}}(\mathsf{LIN}, \mathsf{LIN}) \subseteq S_{\text{И}}(\mathsf{LIN}, \mathsf{CF})$ follows. Hence this part of the table does not change, and since exact classifications are obtained, we have proved the theorem. □

We now show that $S_{\mathsf{Z}}(\mathsf{REG}, \mathcal{F}_2) = S_{\text{И}}(\mathsf{REG}, \mathcal{F}_2)$ also holds (for $\mathcal{F}_2 = \mathsf{LIN}, \mathsf{CF}$), by giving a direct proof.

We start by providing a normal form for splicing systems with regular initial language and rules from a family that is closed under GSM mappings. According to this normal form, every splicing rule is of the form (u, p, q, x), where u and x are strings, while p and q are *symbols*.

This normal form is suggested by the fact that the strings v_1 and u_2 do not appear in the result when a splicing rule (u_1, v_1, u_2, v_2) is applied. We only need the fact that the initial strings have these substrings next to the cutting points. However, the interchange of these two strings causes the fact that $\mathcal{R}_{\mathsf{Z}}(\mathsf{LIN}) \neq \mathcal{R}_{\text{И}}(\mathsf{LIN})$ and $\mathcal{R}_{\mathsf{Z}}(\mathsf{CF}) \neq \mathcal{R}_{\text{И}}(\mathsf{CF})$, as explained in Section 4.1. If we are able to restrict v_1 and u_2 to symbols rather than strings, we do not have this problem: since LIN and CF are closed under GSM mappings, it is possible to interchange the two symbols.

We need some notation: let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton, and let $p \in Q$ and $u \in \Sigma^*$. We use $p\overset{u}{\to}$ to denote the fact that p has an outgoing path with label u in the state transition diagram of $\mathcal{A}$, i.e., $\delta(p, u) \neq \varnothing$. Similarly, we write $\overset{u}{\to}p$ if p has an incoming path with label u, i.e., $p \in \delta(q, u)$ for some $q \in Q$.

Lemma 4.7 *Let $\mathcal{F}$ be family of languages that is closed under* GSM *mappings. For each splicing system $h = (V, L, R)$ of $(\mathsf{REG}, \mathcal{F})$ type an equivalent splicing system $h_1 = (V_1, L_1, R_1)$ of $(\mathsf{REG}, \mathcal{F})$ type can be constructed with $R_1 \subseteq V_1^* \times V_1 \times V_1 \times V_1^*$.*

Proof. *Construction.* Let $\mathcal{A} = (Q, V, \delta, q_0, F)$ be a deterministic finite automaton accepting L, with $Q \cap V = \varnothing$. We may assume that $\mathcal{A}$ is 'reduced', i.e., every state in Q occurs on a path from the initial state to a final state. Define $Q' = \{q' \mid q \in Q\}$ and $Q'' = \{q'' \mid q \in Q\}$. Let $L_{\to p}$ be the language accepted by $\mathcal{A}_{\to p} = (Q, V, \delta, q_0, \{p\})$, and similarly let $L_{p\to}$ be the language accepted by $\mathcal{A}_{p\to} = (Q, V, \delta, p, F)$. Define $h_1 = (V \cup Q' \cup Q'', L_1, R_1)$ as follows:

$$
\begin{aligned}
L_1 &= \bigcup_{p \in Q} ((L_{\to p} \cdot p') \cup (p'' \cdot L_{p\to})) \\
R_1 &= \{(u_1, p', q'', v_2) \mid (u_1, v_1, u_2, v_2) \in R,
\end{aligned}
$$

$$p' \in Q',\ q'' \in Q'',\ p\overset{v_1}{\rightarrow} \text{ and } \overset{u_2}{\rightarrow}q\}$$

Correctness. Since both $L_{\rightarrow p}$ and $L_{p\rightarrow}$ are regular and Q is finite, L_1 is a regular language.

Consider the Z-representation of a splicing rule, $u_1\#v_1\$u_2\#v_2$. The translation of $u_1\#v_1\$u_2\#v_2$ into $u_1\#p'\$q''\#v_2$ can be realized by a GSM mapping, that simulates the transition diagram of $\mathcal{A}$. Such a GSM copies $u_1\#$ to the output, guesses a state p and writes $p'\$$, and then reads without writing until it reaches \$, in the meantime checking that v_1 can be read in $\mathcal{A}$ starting in p. After \$ it does something similar for $u_2\#v_2$. Since $\mathcal{F}$ is closed under GSM mappings we have $Z(R_1) \in \mathcal{F}$.

From the construction above, it is clear that $x = x_1u_1v_1y_1$ and $y = x_2u_2v_2y_2$ are in L, for $x_i, u_i, v_i, y_i \in V^*$, if and only if $x' = x_1u_1 \cdot p' \in L_1$, with p such that $\overset{x_1u_1}{\rightarrow} p$, and $y' = q'' \cdot v_2y_2 \in L_1$, with q such that $q\overset{v_2y_2}{\rightarrow}$. Moreover, $r = (u_1, v_1, u_2, v_2) \in R$ if and only if $r' = (u_1, p', q'', v_2) \in R_1$ with $p\overset{v_1}{\rightarrow}$ and $\overset{u_2}{\rightarrow}q$.

Consequently we have $(x_1u_1 \mid v_1y_1, x_2u_2 \mid v_2y_2) \vdash_r x_1u_1v_2y_2$ if and only if $(x_1u_1 \mid p', q'' \mid v_2y_2) \vdash_{r'} x_1u_1v_2y_2$. □

Hence there exists an effective construction that transforms a splicing system of $(\mathsf{REG}, \mathcal{F})$ type into an equivalent splicing system of $(\mathsf{REG}, \mathcal{F})$ type that is in normal form. Recall that the 'default' string representation is the Z-representation (see page 32). Using a GSM mapping similar to the one used in the proof of Lemma 4.7 we can prove that the lemma also holds when using the И-representation.

Furthermore, the translation of the Z-representation of a rule *in normal form* into the И-representation (i.e., $u\#p\$q\#x \mapsto u\#q\$p\#x$) can also be realized by a GSM mapping. Such a GSM reads and outputs $u\#$, keeps $p\$$ in its finite-state memory, when reading q it outputs $q\$p$, and then reads and outputs $\#x$. Clearly, this also works for the translation of the И- into the Z-representation. Consequently, for a splicing relation R in normal form, $Z(R) \in \mathcal{F}$ if and only if И$(R) \in \mathcal{F}$.

Hence we have the following result, which is applicable for $\mathcal{F} = \mathsf{LIN}, \mathsf{CF}$.

Theorem 4.8 $S_Z(\mathsf{REG}, \mathcal{F}) = S_{И}(\mathsf{REG}, \mathcal{F})$ *whenever* $\mathcal{F}$ *is closed under* GSM *mappings.*

Now, summarizing our results on non-iterated splicing, we have the equality $S_Z(\mathcal{F}_1, \mathcal{F}_2) = S_{И}(\mathcal{F}_1, \mathcal{F}_2)$ for all $\mathcal{F}_1, \mathcal{F}_2$ in the Chomsky hierarchy.

4.3 Families of iterated splicing languages

Similar to the case of non-iterated splicing, we want to know whether the results on iterated splicing change when we use the И-representation instead

of the $\angle$-representation. Since we know that $\mathcal{R}_{\angle}(\mathcal{F}_2) = \mathcal{R}_{\mathsf{N}}(\mathcal{F}_2)$ for $\mathcal{F}_2 = \mathsf{FIN}, \mathsf{REG}, \mathsf{CS}, \mathsf{RE}$, we immediately have the following theorem, for $\mathcal{F}_1$ in the Chomsky hierarchy.

Theorem 4.9 $H_{\angle}(\mathcal{F}_1, \mathcal{F}_2) = H_{\mathsf{N}}(\mathcal{F}_1, \mathcal{F}_2)$ *for* $\mathcal{F}_2 = \mathsf{FIN}, \mathsf{REG}, \mathsf{CS}, \mathsf{RE}$.

For $\mathcal{F}_2 = \mathsf{LIN}, \mathsf{CF}$ we have the following result.

Theorem 4.10 *For* $\mathcal{F}_2 = \mathsf{LIN}, \mathsf{CF}$ *and arbitrary* $\mathcal{F}_1$, $H_{\mathsf{N}}(\mathcal{F}_1, \mathcal{F}_2)$ *has the same upper and lower bounds in the Chomsky hierarchy as* $H_{\angle}(\mathcal{F}_1, \mathcal{F}_2)$.

Proof. We check whether the results used in [HPP97] to fill the LIN and CF columns of Table 3.2 also hold when the N-representation is used. Those results are the following:

(1) $\mathcal{F}_1 \subseteq H_{\angle}(\mathcal{F}_1, \mathcal{F}_2)$ [HPP97, Lemma 3.12],

(2) $H_{\angle}(\mathcal{F}_1, \mathcal{F}_2) \not\subseteq \mathcal{F}_1$ for $\mathcal{F}_1 \in \{\mathsf{REG}, \mathsf{LIN}, \mathsf{CF}, \mathsf{CS}\}$ [HPP97, Lemma 3.13],

(3) $H_{\angle}(\mathsf{FIN}, \mathsf{FIN})$ contains infinite languages [HPP97, discussion in proof of Theorem 3.3],

(4) For all $L \subseteq V^*, L \notin \mathcal{F}_1$ and $c, d \notin V$ we have
$L' = (dc)^* L (dc)^* \cup c(dc)^* L (dc)^* d \notin H_{\angle}(\mathcal{F}_1, \mathcal{F}_2)$ [HPP97, Lemma 3.16],

(5) $H_{\angle}(\mathcal{F}_1, \mathcal{F}_2) \not\subseteq \mathsf{CS}$ for $\mathcal{F}_2 \neq \mathsf{FIN}$ [HPP97, Lemma 3.15].

In the proofs of (1) and (3) the $\angle$-representation of the splicing relation is in FIN, for (4) it is arbitrary (i.e., RE), and for (5) it is in REG. These families are closed under $\mathsf{N}\angle^{-1}$, hence the same results are obtained when we use the N-representation.

In the proof of (2) from a splicing relation R a new splicing relation $R' = \{(u_1, cv_1, u_2c, v_2) \mid (u_1, v_1, u_2, v_2) \in R\}$ is constructed. Such a construction also works when the N-representation is used: $\mathsf{N}(R')$ can be obtained from $\mathsf{N}(R)$ by changing the \$ into $c\$c$. The families in the Chomsky hierarchy are closed under this operation (λ-free homomorphism).

For $\mathcal{F}_1 \neq \mathsf{RE}$, by (1), (2), (3) we have $\mathcal{F}_1 \subset H_{\mathsf{N}}(\mathcal{F}_1, \mathcal{F}_2)$, while from (4) and (5) it follows that the smallest upper bound for $H_{\mathsf{N}}(\mathcal{F}_1, \mathcal{F}_2)$ is RE. By (1) immediately $\mathsf{RE} \subseteq H_{\mathsf{N}}(\mathsf{RE}, \mathcal{F}_2)$. □

Consequently, Table 3.2 does not change when we use the N-representation instead of the $\angle$-representation. Note, however, that we have *not* proved that $H_{\angle}(\mathcal{F}_1, \mathcal{F}_2) = H_{\mathsf{N}}(\mathcal{F}_1, \mathcal{F}_2)$, for $\mathcal{F}_2 = \mathsf{LIN}, \mathsf{CF}$, except for the obvious case in which $\mathcal{F}_1 = \mathsf{RE}$ where upper and lower bound coincide with RE.

4.4 Representations other than Z and И

We have considered one alternative string representation for splicing relations, that separates left and right contexts rather than the two initial strings. However, there are 24 possible representations in the '#\$#'-style (i.e., representations of a splicing rule (u_1, v_1, u_2, v_2) where u_1, v_1, u_2, v_2 are concatenated in any order and separated by #, \$ and # (in this order)), corresponding to the permutations of the four components of the splicing rules. We do not claim that all these permutations have a natural interpretation, but still we believe that the effect of using any of these representations on the classification results given in Tables 3.1 and 3.2 should be determined. In the sequel we discuss the remaining possibilities in a rather informal way. This discussion also holds for string representations that use, for instance, three times the same symbol, or three different symbols to separate the four parts of a splicing rule.

4.4.1 Splicing with FIN, REG, CS or RE rules

For (non-iterated or iterated) splicing with FIN, REG, CS or RE splicing rules, it does not matter which of the '#\$#'-representations is used: for a finite splicing relation obviously each of these representations yields a finite language, and for a REG, CS or RE splicing relation R, an inverse 2DGSM mapping like the one which is used to prove Lemma 4.3 can transform $\mathsf{Z}(R)$ into $\rho(R)$ for each of the other representations ρ in this style. In other words, if ρ is one of these representations, then $\mathcal{R}_{\mathsf{Z}}(\mathcal{F}_2) = \mathcal{R}_\rho(\mathcal{F}_2)$ and consequently $S_{\mathsf{Z}}(\mathcal{F}_1, \mathcal{F}_2) = S_\rho(\mathcal{F}_1, \mathcal{F}_2)$ and $H_{\mathsf{Z}}(\mathcal{F}_1, \mathcal{F}_2) = H_\rho(\mathcal{F}_1, \mathcal{F}_2)$, for $\mathcal{F}_2 = \mathsf{FIN}, \mathsf{REG}, \mathsf{CS}, \mathsf{RE}$.

4.4.2 Splicing with LIN or CF rules

Non-iterated splicing non-REG languages with LIN or CF splicing rules yields the same classification in each one of the '#\$#'-representations, because the results used in [HPP97] to fill the corresponding part of Table 3.1 are easily seen to hold for all string representations ρ in this style, cf. Theorem 4.6. Since in these classifications upper and lower bound coincide, this implies that $S_{\mathsf{Z}}(\mathcal{F}_1, \mathcal{F}_2) = S_\rho(\mathcal{F}_1, \mathcal{F}_2)$, for $\mathcal{F}_1 \neq \mathsf{REG}$ and $\mathcal{F}_2 = \mathsf{LIN}, \mathsf{CF}$.

In the case of non-iterated splicing REG languages with CF splicing rules, we use the normal form of Section 4.2 for the rules, that makes it possible to change the Z-representation $u\#p\$q\#x$ of a splicing rule into the И-representation $u\#q\$p\#x$ by applying a GSM mapping (recall that this is possible only because p and q are symbols). Obviously, there exist GSM mappings that map $u\#p\$q\#x$ into each of the 12 representations in which u precedes x. To see that the other 12 possibilities can also be obtained by operations preserving context-freeness, note that CF is closed under the operation CYCLE, that can move x in front of u [HU79, Exercise 6.4c]. Since, for a language K, CYCLE(K) is defined as $\{yx \mid xy \in K \text{ for some } x \text{ and } y\}$, we need some extra operations to make sure

that, for instance, $u\#p\$q\#x$ is transformed only in $x\#u\$p\#q$. This can be done by first creating $u\#p\$q\#x\diamond$, where $\diamond$ is a new symbol, then CYCLE followed by intersection with an appropriate regular set to obtain $x \diamond u\#p\$q\#$, and finally a GSM mapping to get $x\#u\$p\#q$. Since CF is closed under all these operations, this shows that $S_{\angle}(\mathsf{REG},\mathsf{CF}) = S_\rho(\mathsf{REG},\mathsf{CF})$ for each '#\$#'-representation ρ.

For non-iterated splicing REG languages with LIN rules, however, we give an example that shows that the '#\$#'-representations of a rule (u,v,w,x) in which x precedes u are *not* equivalent to the $\angle$-representation.

Example 4.2 Consider a splicing system $h = (\{a,b,c,d\}, L, R)$ with

$$\begin{aligned} L &= c \cdot \{a,b\}^* \cdot d \\ R &= \{(cb^j, d, c, a^n b^i d) \mid i,j,n \geq 1 \text{ and } i+j=n\}. \end{aligned}$$

Then the 'reverse' representation of R (i.e., the representation that transforms (u,v,w,x) into $x\#w\$v\#u$) is

$$R_r = \{a^n b^i d\#c\$d\#cb^j \mid i,j,n \geq 1 \text{ and } i+j=n\},$$

which is a linear language. The non-iterated splicing language generated by h,

$$\sigma(h) = \{cb^j a^n b^i d \mid i,j,n \geq 1 \text{ and } i+j=n\},$$

is *not* in LIN (by the pumping lemma for linear languages [HU79, Exercise 6.11]). Since $S_{\angle}(\mathsf{REG},\mathsf{LIN}) \subset \mathsf{LIN}$, clearly this 'reverse' representation is not equivalent to the $\angle$-representation. □

Of course, all representations ρ in which u precedes x *are* equivalent to the $\angle$-representation, by using the same arguments as in the CF case. So for those representations we have $S_{\angle}(\mathsf{REG},\mathsf{LIN}) = S_\rho(\mathsf{REG},\mathsf{LIN})$.

For iterated splicing, it is easy to see that the results used in [HPP97] to fill the LIN and CF columns of Table 3.2 hold for every '#\$#'-representation, by observations as in the proof of Theorem 4.10. Hence this part of that table does not change either.

4.5 Summary

Summarizing the results of this chapter, we see that for non-iterated splicing families all '#\$#'-representations are equivalent, except for the twelve cases mentioned above, where a REG initial language is spliced using LIN rules.

For iterated splicing, however, we have seen that those representations are only equivalent when splicing with FIN, REG, CS or RE rules; in the case of LIN or CF rules we know that the classifications (i.e., the upper and lower bounds) in Table 3.2 do not change, but we do not yet know whether or not all families of iterated splicing languages stay the same.

Chapter 5

Non-iterated splicing with regular rules

The first two columns of Table 3.1 show that $S(\mathcal{F}, \mathsf{FIN})$ equals $S(\mathcal{F}, \mathsf{REG})$ for all $\mathcal{F}$ except LIN, for which it is only known that they have the same upper and lower bounds. We give a direct proof of the equalities $S(\mathcal{F}, \mathsf{FIN}) = S(\mathcal{F}, \mathsf{REG})$ for all Chomsky families, and moreover we prove that $S(\mathcal{F}, \mathsf{FIN}) = \mathcal{F} \oplus \mathcal{F}$, for each Chomsky family $\mathcal{F}$ except CS. This yields the missing characterizations in Table 3.1: $S(\mathsf{LIN}, \mathsf{FIN}) = S(\mathsf{LIN}, \mathsf{REG}) = \mathsf{LIN} \oplus \mathsf{LIN}$.

We try to replace regular rule sets by finite rule sets for the four kinds of restricted splicing that we consider as well.

5.1 Unrestricted splicing

We first give a characterization of $S(\mathsf{LIN}, \mathsf{FIN})$ in terms of LIN. We do this in a general setting: we give a characterization of $S(\mathcal{F}, \mathsf{FIN})$ in terms of $\mathcal{F}$, for each family $\mathcal{F}$ that is closed under GSM mappings, union and concatenation with symbols.

For a language family $\mathcal{F}$ we use $\mathcal{F} \oplus \mathcal{F}$ to denote finite unions of elements of $\mathcal{F}^2$, i.e., languages of the form $K_1 \cdot L_1 \cup \ldots \cup K_n \cdot L_n$, $n \geq 0$, with $K_i, L_i \in \mathcal{F}$. If we assume that $\{\lambda\}$ and $\varnothing$ are elements of $\mathcal{F}$, then $\mathcal{F} \oplus \mathcal{F}$ equals $\mathcal{F}$ if and only if $\mathcal{F}$ is closed under union and concatenation. Hence $\mathcal{F} \oplus \mathcal{F} = \mathcal{F}$ for each Chomsky family except LIN, which is not closed under concatenation.

Lemma 5.1 *Let $\mathcal{F}$ be a family of languages closed under* GSM *mappings, union and concatenation with symbols. Then $S(\mathcal{F}, \mathsf{FIN}) = \mathcal{F} \oplus \mathcal{F}$.*

Proof. First we show that $S(\mathcal{F}, \mathsf{FIN}) \subseteq \mathcal{F} \oplus \mathcal{F}$. Let $h = (V, L, R)$ be a splicing system with a finite number of rules and with $L \in \mathcal{F}$.

Consider the rule $r = (u_1, v_1, u_2, v_2)$. When r is applied to strings $x_1 u_1 v_1 y_1$ and $x_2 u_2 v_2 y_2$, then only the substrings $x_1 u_1$ and $v_2 y_2$ are visible in the resulting

string $x_1u_1v_2y_2$. We define two languages derived from the initial language following this observation: let $L_{\langle r} = \{xu_1 \mid xu_1v_1y \in L \text{ for some } x, y \in V^*\}$, and let $L_{r\rangle} = \{v_2y \mid xu_2v_2y \in L, \text{ for some } x, y \in V^*\}$.

Observe that both $L_{\langle r}$ and $L_{r\rangle}$ can be obtained from L by a GSM mapping that, in the case of $L_{\langle r}$, reads and outputs x, guesses that it can start reading u_1v_1 for the splicing rule $r = (u_1, v_1, u_2, v_2)$, reads and outputs u_1, and reads the rest of the input without copying it to the output, while checking that it starts with v_1. A similar GSM can construct $L_{r\rangle}$, and consequently these languages are in $\mathcal{F}$. Clearly, $\sigma(h) = \bigcup_{r\in R} L_{\langle r}L_{r\rangle}$, and since there is only a finite number of splicing rules this means that $\sigma(h) \in \mathcal{F} \oplus \mathcal{F}$.

Second, we show that $\mathcal{F} \oplus \mathcal{F} \subseteq S(\mathcal{F}, \mathsf{FIN})$. Consider $K_1 \cdot L_1 \cup \ldots \cup K_n \cdot L_n$ with $K_i, L_i \subseteq V^*$ in $\mathcal{F}$, for some alphabet V and $n \geq 0$. This union is obtained by splicing the initial language $\bigcup_{i=1}^n K_ic_i \cup \bigcup_{i=1}^n c_i'L_i$ with rules $(\lambda, c_i, c_i', \lambda)$, $i = 1, \ldots, n$, where the c_i, c_i' are new symbols. Since $\mathcal{F}$ is closed under union and concatenation with symbols, the initial language belongs to $\mathcal{F}$. □

This lemma is applicable for the Chomsky families FIN, REG, CF, RE and LIN. For the first four of these families it gives the known characterizations $S(\mathcal{F}, \mathsf{FIN}) = \mathcal{F} \oplus \mathcal{F} = \mathcal{F}$. The equality $S(\mathsf{LIN}, \mathsf{FIN}) = \mathsf{LIN} \oplus \mathsf{LIN}$, however, appears to be new, although the family $\mathsf{LIN} \oplus \mathsf{LIN}$ is mentioned in the proof of Theorem 3 of [Pău96a], where it is demonstrated that $S(\mathsf{LIN}, \mathsf{REG})$ is strictly included in CF. In fact, in that proof the inclusion $S(\mathsf{LIN}, \mathsf{REG}) \subseteq \mathsf{LIN} \oplus \mathsf{LIN}$ is proved. Together with our characterization of $S(\mathsf{LIN}, \mathsf{FIN})$ and the fact that $S(\mathsf{LIN}, \mathsf{FIN}) \subseteq S(\mathsf{LIN}, \mathsf{REG})$ this gives the following result, which gives a negative answer to a question from [Pău96a]: is the inclusion $S(\mathsf{LIN}, \mathsf{FIN}) \subseteq S(\mathsf{LIN}, \mathsf{REG})$ proper?

Theorem 5.2 $S(\mathsf{LIN}, \mathsf{FIN}) = S(\mathsf{LIN}, \mathsf{REG}) = \mathsf{LIN} \oplus \mathsf{LIN}$.

Note that indeed $\mathsf{LIN} \subset \mathsf{LIN} \oplus \mathsf{LIN} \subset \mathsf{CF}$: the first inclusion follows from the fact that LIN contains the language $\{\lambda\}$, and it is a proper inclusion since LIN is not closed under concatenation. The second inclusion is true because $\mathsf{LIN} \subseteq \mathsf{CF}$ and CF is closed under concatenation and union; it is proper since languages in $\mathsf{LIN} \oplus \mathsf{LIN}$ have *index* at most two, while context-free languages can have arbitrarily large index (for details, see the proof from [Pău96a] mentioned above).

We can extend Theorem 5.2 to language families that are closed under shuffle with symbols and intersection with regular sets, by refining the proof of Lemma 5.1. In other words, when using an initial language from such a family, regular sets of splicing rules are equivalent to finite sets of splicing rules. We use a technique that is also used in [Pău96a, Theorem 5], where the cutting points in words from the initial language are marked with new symbols in order to reduce a finite rule set to a rule set with radius 1. We show how this technique can be applied to (infinite) regular rule sets as well.

Theorem 5.3 *Let $\mathcal{F}$ be a family of languages closed under shuffle with symbols and intersection with regular sets. Then $S(\mathcal{F}, \mathsf{FIN}) = S(\mathcal{F}, \mathsf{REG})$.*

Proof. The inclusion $S(\mathcal{F}, \mathsf{FIN}) \subseteq S(\mathcal{F}, \mathsf{REG})$ is clear. For the converse inclusion we use the following construction.

Construction. Let $h = (V, L, R)$ be a splicing system with $L \in \mathcal{F}$ and $\mathcal{Z}(R) \in \mathsf{REG}$. Let $\mathcal{A} = (Q, V \cup \{\#, \$\}, \delta, q_0, F)$ be a deterministic finite automaton with $L(\mathcal{A}) = \mathcal{Z}(R)$, $Q \cap (V \cup \{\#, \$\}) = \varnothing$ and $\#, \$ \notin V$. We construct a splicing system $h' = (V \cup Q, L', R')$ with $L' \in \mathcal{F}$, R' finite and $\sigma(h') = \sigma(h)$ as follows.

$$\begin{aligned}
L' &= \{xupvy \mid x,u,v,y \in V^*,\, xuvy \in L \text{ and } \delta(q_0, u\#v) = p\,\} \\
&\quad \cup \\
&\quad \{xuqvy \mid x,u,v,y \in V^*,\, xuvy \in L \text{ and } \delta(q, u\#v) \in F\,\} \\
R' &= \{(\lambda, p, q, \lambda) \mid \delta(p, \$) = q\}
\end{aligned}$$

Correctness. First, observe that L' can be constructed from L as follows: $L' = (L \dagger Q) \cap K$, where $\dagger$ denotes the shuffle operation and $K = \{xupvy \mid x,u,v,y \in V^*$ and either $\delta(q_0, u\#v) = p$ or $\delta(p, u\#v) \in F\}$. Note that the language K can be constructed from $\mathcal{Z}(R)$ by a non-deterministic non-erasing FST mapping that uses $\mathcal{A}$. For example, when computing xu_1pv_1y from $u_1\#v_1\$u_2\#v_2$ the FST first non-deterministically generates x and then simulates $\mathcal{A}$ on the segment u_1 of its input, while copying its input to the output. At the end of u_1, it reads $\#$ and writes p to the output, where p is non-deterministically guessed. The FST now reads v_1 and copies it to the output, while checking whether state p is reached after reading v_1 in $\mathcal{A}$. After this it reads $\$u_2\#v_2$ without copying it to the output, and then it non-deterministically generates y. To compute xu_2qv_2y from $u_1\#v_1\$u_2\#v_2$ a similar procedure is followed. Also note that, because REG is closed under non-erasing FST mappings, K is regular. Since $\mathcal{F}$ is closed under shuffle with symbols and intersection with regular sets, L' is in $\mathcal{F}$.

Second, it is clear that R' is finite, since $\mathcal{A}$ has only finitely many transitions.

Third, we prove that $\sigma(h) \subseteq \sigma(h')$. Let $x = x_1u_1v_1y_1 \in L$, $y = x_2u_2v_2y_2 \in L$ and $r = u_1\#v_1\$u_2\#v_2 \in \mathcal{Z}(R)$ be such that $(x, y) \vdash_r x_1u_1v_2y_2$. Then there are $p, q \in Q$ such that $\delta(q_0, u_1\#v_1) = p$, $\delta(p, \$) = q$ and $\delta(q, u_2\#v_2) \in F$. Consequently there are $x' = x_1u_1pv_1y_1 \in L'$, $y' = x_2u_2qv_2y_2 \in L'$ and $r' = (\lambda, p, q, \lambda) \in R'$ with $(x', y') \vdash_{r'} x_1u_1v_2y_2$.

Finally, we prove that $\sigma(h') \subseteq \sigma(h)$. Let $x' = upv \in L'$, $y' = wqz \in L'$, $r' = (\lambda, p, q, \lambda) \in R'$ and $(x', y') \vdash_{r'} uz$. Then it must be that $u = x_1u_1$ and $v = v_1y_1$, for some $x_1, u_1, v_1, y_1 \in V^*$ such that $\delta(q_0, u_1\#v_1) = p$, and $w = x_2u_2$, $z = v_2y_2$, for some $x_2, u_2, v_2, y_2 \in V^*$ such that $\delta(q, u_2\#v_2) \in F$. Because $(\lambda, p, q, \lambda) \in R'$, it also holds that $\delta(p, \$) = q$. Consequently there is a rule $r = u_1\#v_1\$u_2\#v_2 \in \mathcal{Z}(R)$. Moreover, by the construction of L', there must

be $x = x_1u_1v_1y_1 \in L$ and $y = x_2u_2v_2y_2 \in L$ such that $(x, y) \vdash_r x_1u_1v_2y_2 = uz$. Hence $\sigma(h) = \sigma(h')$. □

Note that every rule (λ, p, q, λ) in R' corresponds to a (regular) set of rules $\{ u_1\#v_1\$u_2\#v_2 \mid \delta(q_0, u_1\#v_1) = p \text{ and } \delta(q, u_2\#v_2) \in F \} \subseteq Z(R)$.

Again, for all Chomsky families except LIN the last result is implicit in Table 3.1. Here it is obtained through direct construction.

Note that we cannot replace the shuffle with symbols and the intersection with regular sets by a GSM mapping, because it may be that the empty word is one of the terms of the splicing and GSM's are not allowed to write a non-empty string while reading λ.

Also note that the construction in the proof of Theorem 5.3 gives an affirmative answer to another question posed in [Pău96a]: can each language in $S(\mathcal{F}, \mathsf{REG})$ be represented in a 'simple' way starting from languages in $S(\mathcal{F}, [1])$? Indeed, whenever $\mathcal{F}$ is closed under shuffle with symbols and intersection with regular sets, then $S(\mathcal{F}, \mathsf{REG}) = S(\mathcal{F}, [1])$.

5.2 Same-length splicing

A closer look at the proof of Theorem 5.3 reveals that it also works for non-iterated splicing in same-length mode, where the splicing of x and y is only allowed if $|x| = |y|$.

Theorem 5.4 *Let $\mathcal{F}$ be a family of languages closed under shuffle with symbols and intersection with a regular set. Then $S_{sl}(\mathcal{F}, \mathsf{FIN}) = S_{sl}(\mathcal{F}, \mathsf{REG})$.*

Proof. It is clear that in the construction used in the proof of Theorem 5.3 it holds that $|x'| = |x| + 1$ and $|y'| = |y| + 1$. Thus we have $|x| = |y|$ if and only if $|x'| = |y'|$ and consequently $\sigma_{sl}(h) = \sigma_{sl}(h')$. □

5.3 Self splicing

In the case of self splicing, where a string is spliced with itself, both splicing sites are found in that same string. Hence the strategy of marking the two cutting points of a splicing rule (u_1, v_1, u_2, v_2) in the input string with new symbols (say p and q) does not work: if the second splicing site u_2v_2 occurs before the first splicing site u_1v_1, then self splicing $xu_2\, q\, v_2yu_1\, p\, v_1z$ with the new splicing rule (λ, p, q, λ) gives a word that still contains the auxiliary symbols q and p. We will explain that, at least for self splicing with a regular initial language, this is not caused by this particular construction: the family $S_{sf}(\mathsf{REG}, \mathsf{FIN})$ is strictly contained in the family $S_{sf}(\mathsf{REG}, \mathsf{REG})$. This can be proved using the language from the following example.

Example 5.1 Consider the splicing system $h = (\{a, b\}, L, R)$ defined by

$$\begin{aligned} L &= a^*ba^*ba^* \in \mathsf{REG} \\ \mathcal{Z}(R) &= ba^* \# b \$ \# ba^*b \in \mathsf{REG} \end{aligned}$$

When splicing in self splicing mode, the only rule that is applicable to $a^\ell ba^m ba^n$ is $ba^m\#b\$\#ba^m b$, which gives as result $a^\ell ba^m ba^m ba^n$. Hence

$$\sigma_{sf}(h) = \{a^\ell ba^m ba^m ba^n \mid \ell, m, n \geq 0\} \in S_{sf}(\mathsf{REG}, \mathsf{REG}).$$

□

Let (u_1, v_1, u_2, v_2) be a splicing rule. Since in the case of self splicing both splicing sites can be found in the same string, the input string can be written as xyz, where either $u_1 \in \mathrm{Suf}(x)$, $v_1 \in \mathrm{Pref}(yz)$, $u_2 \in \mathrm{Suf}(xy)$, $v_2 \in \mathrm{Pref}(z)$ and the result of self splicing is xz, or $u_1 \in \mathrm{Suf}(xy)$, $v_1 \in \mathrm{Pref}(z)$, $u_2 \in \mathrm{Suf}(x)$, $v_2 \in \mathrm{Pref}(yz)$ and the result of self splicing is $xyyz$. Note that the situation where the two cutting points coincide is covered by both the first and the second case ($y = \lambda$ and the result of the splicing equals the input string).

We will show that $K = \{a^\ell ba^m ba^m ba^n \mid \ell, m, n \geq 0\}$ cannot be created by self splicing a regular initial language using only a finite number of rules. The idea behind the proof is similar to that behind the proof of $a^*ba^*ba^* \notin H(\mathsf{FIN}, \mathsf{FIN})$ (see Example 3.5): with a finite number of rules either one cannot control the number of b's in the resulting string, or one cannot get the desired numbers of a's in between the b's.

Theorem 5.5 $S_{sf}(\mathsf{REG}, \mathsf{FIN}) \subset S_{sf}(\mathsf{REG}, \mathsf{REG})$

Proof. The inclusion $S_{sf}(\mathsf{REG}, \mathsf{FIN}) \subseteq S_{sf}(\mathsf{REG}, \mathsf{REG})$ follows from the definitions. We show that the language $K = \{a^\ell ba^m ba^m ba^n \mid \ell, m, n \geq 0\}$ from the previous example is not in $S_{sf}(\mathsf{REG}, \mathsf{FIN})$.

Suppose that $h = (V, L, R)$ is a splicing system with $L \in \mathsf{REG}$, R finite and $\sigma_{sf}(h) = K$. Since R is finite we can define $k = \max\{|u_i| \mid i = 1, \ldots, 4$ and $(u_1, u_2, u_3, u_4) \in R\}$. Let w be a word in $\sigma_{sf}(h)$.

First, observe that if $w = xz$, where $(xyz, xyz) \vdash_r^{sf} xz$ for a word $xyz \in L$, a splicing rule $r = (u_1, v_1, u_2, v_2) \in R$ with $u_1 \in \mathrm{Suf}(x)$, $v_1 \in \mathrm{Pref}(yz)$, $u_2 \in \mathrm{Suf}(xy)$ and $v_2 \in \mathrm{Pref}(z)$, then this splicing can be simulated by a GSM that, on input xyz, reads and outputs x, reads y without generating output, and reads and outputs z, while at the same time checking that u_1, v_1, u_2, v_2 appear in the right places. Since the initial language is regular and since REG is closed under GSM mappings, this way of splicing leads to a regular language (let us call this language K_{xz}). However, the language K is in $\mathsf{CF} - \mathsf{REG}$, hence it cannot be defined with only this kind of self splicing. In fact, to create K we need an infinite number of self splicings of the other kind, i.e., there are

infinitely many words $w \in \sigma_{sf}(h) - K_{\rm xz}$ with $w = xyyz$ for some $xyz \in L$ such that $(xyz, xyz) \vdash_r^{sf} xyyz$, where $r = (u_1, v_1, u_2, v_2) \in R$ and $u_1 \in \text{Suf}\,(xy)$, $v_1 \in \text{Pref}\,(z)$, $u_2 \in \text{Suf}\,(x)$, $v_2 \in \text{Pref}\,(yz)$. This is caused by the fact that $K - K_{\rm xz}$ must be infinite, since if it were finite, then $(K - K_{\rm xz}) \cup K_{\rm xz} = K$ would be regular.

Since $K_{\rm xz} \subset K$ is regular, it must hold that $K_{\rm xz} \subseteq \{a^\ell ba^i ba^i ba^n \mid \ell, n \geq 0$ and $0 \leq i \leq m\}$ for a certain $m \geq 0$. Then $K - K_{\rm xz}$ should contain at least all words of the form $a^\ell ba^i ba^i ba^n$ with $\ell, n \geq 0$ and $i > m$.

Hence there exists a $w = a^\ell ba^m ba^m ba^n \in K$ with $\ell, m, n > 2k$, and such that $w = xyyz$ as described above. Then either $y = a^p$ with $p \geq 1$ or $y = a^i ba^j$ with $i, j \geq 0$, because w contains exactly three b's and y occurs twice in w.

If $y = a^p$, then the two (adjacent) copies of y occur inside one of the four groups of consecutive a's in w. We discuss one of these cases in detail, the other cases can be handled in a similar way. Suppose that $x = a^q$, for some $q \geq 0$, and that $z = a^{\ell-q-2p}ba^m ba^m ba^n$ (hence $xyz = a^{\ell-p}ba^m ba^m ba^n$). Let $r = (u_1, v_1, u_2, v_2) \in R$ be such that $(xyz, xyz) \vdash_r^{sf} xyyz = w$. Then $u_1 = a^{\ell_1}$ and $u_2 = a^{\ell_2}$, for $0 \leq \ell_1, \ell_2 \leq k$, while for v_1 and v_2 four combinations are possible: either $v_1 = a^{n_1}$ with $0 \leq n_1 \leq k$, or $v_1 = a^{n_1}ba^{n_2}$ for $1 \leq n_1 + 1 + n_2 \leq k$, and either $v_2 = a^{n_3}$ for $0 \leq n_3 \leq k$, or $v_2 = a^{n_3}ba^{n_4}$ for $1 \leq n_3 + 1 + n_4 \leq k$. Now also the following self splicing of xyz using r is possible:

$$(a^{\ell-p}ba^m ba^{m-\ell_1-n_1}a^{\ell_1} \mid a^{n_1}ba^n \,,\, a^{\ell-p-\ell_2-n_3}a^{\ell_2} \mid a^{n_3}ba^m ba^m ba^n)$$

$$\vdash_r^{sf} \; a^{\ell-p}ba^m ba^{m-n_1+n_3}ba^m ba^m ba^n$$

for which the resulting string contains too many b's.

If $y = a^i ba^j$ with $i, j \geq 0$, then either $x = a^{\ell-i}$ and $z = a^{m-j}ba^n$, or $x = a^\ell ba^{m-i}$ and $z = a^{n-j}$. In both cases the input word can be reconstructed as $xyz = a^\ell ba^m ba^n$. Again we only discuss one of the two cases, the other one is analogous. In the second case, $v_1 = a^{n_1}$ for $0 \leq n_1 \leq k$, and either $u_1 = a^{\ell_1}$ for $0 \leq \ell_1 \leq k$ or $u_1 = a^{\ell_2}ba^{\ell_1}$ for $1 \leq \ell_2 + 1 + \ell_1 \leq k$. For the second splicing site, it is enough to know that it cuts the input word between $a^\ell ba^{m-i}$ and $a^{m-j}ba^n$ (note that $m = i + j$). Now the following self splicing is possible:

$$(a^\ell ba^{\ell_1} \mid a^{n_1}a^{m-\ell_1-n_1}ba^n \,,\, a^\ell ba^{m-i} \mid a^{m-j}ba^n) \;\vdash_r^{sf}\; a^\ell ba^{m-j+\ell_1}ba^n$$

for which the resulting string has less than three b's and thus does not belong to K.

Hence there is no way of choosing x, y, z such that exactly each string in K is the result of self splicing xyz using a rule from the finite set of splicing rules R. Consequently $K \in S_{sf}(\mathsf{REG}, \mathsf{REG}) - S_{sf}(\mathsf{REG}, \mathsf{FIN})$. □

5.4 Length-decreasing splicing

When splicing in length-decreasing mode, the resulting string should be strictly shorter than the input strings. For the input strings $x_1u_1v_1y_1$ and $x_2u_2v_2y_2$, spliced using the rule (u_1, v_1, u_2, v_2), and the resulting string $x_1u_1v_2y_2$ this condition can also be written as $|x_1u_1| < |x_2u_2|$ and $|v_2y_2| < |v_1y_1|$. Therefore the construction that we used in the case of unrestricted (and same-length) splicing to replace a regular set of splicing rules by a finite one, i.e., marking the cutting points in the input strings with extra, new symbols and cutting those off by splicing, does not work here: the relation between the lengths of the input strings and the length of the resulting string is disturbed. Fortunately this can easily be repaired: like we have seen before, when $x_1u_1v_1y_1$ and $x_2u_2v_2y_2$ are spliced using the rule (u_1, v_1, u_2, v_2), the substrings v_1y_1 and x_2u_2 do not appear in the result. Therefore we can *replace* the first letter of v_1y_1 and the last letter of x_2u_2 by the new symbol and again remove these new symbols by splicing. Indeed, these two letters exist, because when splicing in length-decreasing mode it has to be that $|v_1y_1| > 0$ and $|x_2u_2| > 0$, since otherwise it can never be that $|v_1y_1| > |v_2y_2|$ and $|x_2u_2| > |x_1u_1|$, respectively. Therefore, the replacement construction described above works for every pair of words from the initial language that can splice in length-decreasing mode.

Since only the lengths of v_1y_1 and x_2u_2 are important, we simplify the construction by replacing each symbol in these two strings by ι, where ι is a new symbol.

Note that a consequence of the above discussion is that the empty word cannot be a term in any length-decreasing splicing. Therefore, contrary to the situation in Theorem 5.3, here we can use a (non-erasing) GSM to perform the construction proposed above.

Theorem 5.6 *Let $\mathcal{F}$ be a family of languages closed under non-erasing* GSM *mappings. Then $S_{de}(\mathcal{F}, \mathsf{FIN}) = S_{de}(\mathcal{F}, \mathsf{REG})$.*

Proof. It is clear that $S_{de}(\mathcal{F}, \mathsf{FIN}) \subseteq S_{de}(\mathcal{F}, \mathsf{REG})$. For the converse inclusion we use the following construction.

Construction. For a splicing system $h = (V, L, R)$ with $L \in \mathcal{F}$ and $\angle(R) = L(\mathcal{A})$ for a deterministic finite automaton $\mathcal{A} = (Q, V \cup \{\#, \$\}, \delta, q_0, F)$ with $Q \cap (V \cup \{\#, \$\}) = \varnothing$ and $\#, \$ \notin V$, we construct a splicing system $h' = (V \cup Q, L', R')$ as follows:

$$\begin{aligned} L' &= \{\, xup\iota^{|vy|-1} \mid xuvy \in L,\ vy \neq \lambda \text{ and } \delta(q_0, u\#v) = p \,\} \\ &\quad \cup \\ &\quad \{\, \iota^{|xu|-1}qvy \mid xuvy \in L,\ xu \neq \lambda \text{ and } \delta(q, u\#v) \in F \,\} \\ R' &= \{\, (\lambda, p, q, \lambda) \mid \delta(p, \$) = q\} \end{aligned}$$

Correctness. It is clear that R' is finite.

Furthermore, the new initial language L' can be constructed from L by a non-deterministic non-erasing GSM mapping that uses $\mathcal{A}$. For example, when computing $xup\iota^{|vy|-1}$ from $xuvy$ the GSM guesses the start of the segment u on its input and simulates $\mathcal{A}$ on this segment (all the time copying the input to the output). At the end of u, it simulates the step of $\mathcal{A}$ on $\#$ and writes p to the output, where p is non-deterministically guessed. The GSM now continues to simulate $\mathcal{A}$ on the input, writing a symbol ι to the output for all but one of the remaining symbols of the input, while checking whether state p is reached after reading v in $\mathcal{A}$. Some care has to be taken here. By definition, a GSM cannot use a λ-transition to simulate $\mathcal{A}$ on the additional symbol $\#$ that is not part of the input. As a solution, the GSM may keep in its finite-state memory the values of both $\delta(q_0, u')$ and $\delta(q_0, u'\#)$ for the prefix u' of u that has been read. Since $\mathcal{F}$ is closed under non-erasing GSM mappings, L' is in $\mathcal{F}$.

Moreover, from the discussion above and the similarity of this construction to the construction used in the proof of Theorem 5.3, it is clear that $\sigma_{de}(h') = \sigma_{de}(h)$. □

5.5 Length-increasing splicing

Two strings $x_1u_1v_1y_1$ and $x_2u_2v_2y_2$ can only yield the string $x_1u_1v_2y_2$ as the result of splicing in length-increasing mode using the rule (u_1, v_1, u_2, v_2) if $|x_1u_1v_2y_2| > |x_iu_iv_iy_i|$ for $i = 1, 2$. These two requirements can be simplified to $|x_1u_1| > |x_2u_2|$ and $|v_2y_2| > |v_1y_1|$, which immediately implies that it always must be that $|x_1u_1| > 0$ and $|v_2y_2| > 0$. Note that this means that the shortest words that can be made by length-increasing splicing are of length two, and the only way to create such a word, say ab, where a and b are symbols, is $(a \mid , \mid b) \vdash_r^{in} ab$, with $r = (c_1, \lambda, \lambda, c_2)$ for $c_1 \in \{\lambda, a\}$ and $c_2 \in \{\lambda, b\}$. This consequence of the requirements for splicing in length-increasing mode causes the fact that there are finite languages (apart from finite languages containing words of length one) that are not in $S_{in}(\mathcal{F}_1, \mathcal{F}_2)$ for all $\mathcal{F}_1, \mathcal{F}_2$, as explained by the following example.

Example 5.2 We show that $\{bb, bab\} \notin S_{in}(\mathsf{RE}, \mathsf{RE})$.
Suppose that $\{bb, bab\} = \sigma_{in}(h)$ for a splicing system $h = (V, L, R)$ with $a, b \in V$ and arbitrary L and R.

First, observe that, as described above, the only possibility to create bb by the length-increasing splicing of two initial words is the following: $(\, b \mid , \mid b \,) \vdash_r^{in} bb$, with $r \in \{\, (c_1, \lambda, \lambda, c_2) \mid c_1, c_2 \in \{\lambda, b\}\}$. Note that this means that the word b should be in L.

Second, to create bab we need a splicing of the form $(\, ba \mid , \ c \mid b \,) \vdash^{in} bab$ or $(\, b \mid c \, , \mid ab \,) \vdash^{in} bab$, where $c \in V \cup \{\lambda\}$. Thus it must be that $ba \in L$ or

$ab \in L$. Then, using the rule r described above, we have $(b \mid , \mid ba) \vdash_r^{in} bba$ or $(ab \mid , \mid b) \vdash_r^{in} abb$. Both these words are not in $\{bb, bab\}$. $\square$

In a similar way it can be proved that $\{bab\} \notin S_{in}(\mathsf{RE}, \mathsf{RE})$. Note, however, that $\{bb\} = \sigma_{in}(h)$ for $h = (\{b\}, \{b\}, \{(b, \lambda, \lambda, b)\}$. Anyway, small words seem to cause problems when splicing in length-increasing mode. Indeed, any regular language that does *not* contain words of length three and smaller is in $S_{in}(\mathsf{REG}, \mathsf{FIN})$, as demonstrated by the following example.

Example 5.3 Let $K \in \mathsf{REG}$, with $K \subseteq \Sigma^*$ for an alphabet Σ. Now let $h = (V, L, R)$ be the splicing system defined by

$$\begin{aligned} V &= \Sigma \cup \{\langle ab \rangle, [ab] \mid a, b \in \Sigma\} \\ L &= \{w \cdot \langle ab \rangle \mid wab \in K \text{ for some } a, b \in \Sigma\} \cup \{[ab] \cdot ab \mid a, b \in \Sigma\} \\ R &= \{(\lambda, \langle ab \rangle, [ab], \lambda) \mid a, b \in \Sigma\} \end{aligned}$$

It is easy to construct a GSM that transforms K into $\{w \cdot \langle ab \rangle \mid wab \in K$ for some $a, b \in \Sigma\}$, and since REG is closed under GSM mappings and under union with finite sets we have $L \in \mathsf{REG}$. Then the only splicings possible are of the form $(w \mid \langle ab \rangle \,,\, [ab] \mid ab) \vdash wab \in K$, which is in length-increasing mode if and only if $|w| > 1$. Clearly $\sigma_{in}(h)$ consists of all words of K that have length at least four, thus for each regular language M that consists of words with length at least four we have $M \in S_{in}(\mathsf{REG}, \mathsf{FIN})$. $\square$

Contrary to length-decreasing splicing, for length-increasing splicing it may be that $|v_1 y_1| = 0$ or $|x_2 u_2| = 0$, thus the replacement construction we used there to reduce the regular set of rules to a finite one will not always work here. We did not succeed in finding a different construction that does always work, but we can adapt the replacement construction in such a way that the only words the new splicing system (with finite rule set) cannot create by splicing in length-increasing mode are words of length two or three. In view of the above examples this seems a reasonable restriction.

Theorem 5.7 *Let $\mathcal{F}$ be a family of languages closed under non-erasing* GSM *mappings. Then for every language K in $S_{in}(\mathcal{F}, \mathsf{REG})$ the language $K|_{>3}$ is in $S_{in}(\mathcal{F}, \mathsf{FIN})$.*

Proof. Let $h = (V, L, R)$ be a splicing system with $L \in \mathcal{F}$ and $\angle(R) = L(\mathcal{A})$ for a deterministic finite automaton $\mathcal{A} = (Q, V \cup \{\#, \$\}, \delta, q_0, F)$ with $Q \cap (V \cup \{\#, \$\}) = \varnothing$ and $\#, \$ \notin V$. We construct a splicing system $h'' = (V'', L'', R'')$ with finite rule set that defines a language 'almost' equal to $\sigma_{in}(h)$ as follows.

First attempt. We start by defining a splicing system $h' = (V \cup Q, L', R')$ with

$$\begin{aligned} L' &= \{x u\, p\, \iota^{|vy|-1} \mid xuvy \in L,\ |vy| \geq 1 \text{ and } \delta(q_0, u\#v) = p\} \ \cup \\ &\quad \{x u\, p \mid xu \in L \text{ and } \delta(q_0, u\#) = p\} \ \cup \\ &\quad \{\iota^{|xu|-1} q\, vy \mid xuvy \in L,\ |xu| \geq 1 \text{ and } \delta(q, u\#v) \in F\} \ \cup \\ &\quad \{q\, vy \mid vy \in L \text{ and } \delta(q, \#v) \in F\} \\ R' &= \{(\lambda, p, q, \lambda) \mid \delta(p, \$) = q\} \end{aligned}$$

As before, L' can be constructed from L by a non-deterministic non-erasing GSM mapping, and R' is finite.

It is easy to understand that $\sigma_{in}(h') \subseteq \sigma_{in}(h)$, following the construction of L' and R'. If $x' = x_1 u_1 p \iota^k$ and $y' = \iota^\ell q v_2 y_2$ in L', for suitable k and ℓ, splice in increasing mode to give $z = x_1 u_1 v_2 y_2$ using the rule $r' = (\lambda, p, q, \lambda)$ in R', then there are strings $x = x_1 u_1 v_1 y_1$ and $y = x_2 u_2 v_2 y_2$ in L that splice to give again z using rule $r = (u_1, v_1, u_2, v_2)$ in R. By construction ($|x| = |x'|$, or $|x| = |x'| - 1$ when $|v_1 y_1| = 0$) we know that $|x| \leq |x'|$, thus $|x'| < |z|$ implies $|x| < |z|$. Mutatis mutandis, this argument is also valid for y and y', so x and y splice in length-increasing mode as well.

The reverse inclusion $\sigma_{in}(h) \subseteq \sigma_{in}(h')$ in general is not true: assume that $z \in \sigma_{in}(h)$, for $x = x_1 u_1 v_1 y_1 \in L$, $y = x_2 u_2 v_2 y_2 \in L$ and $r = (u_1, v_1, u_2, v_2) \in R$ with $(x, y) \vdash_r^{in} x_1 u_1 v_2 y_2 = z$, i.e., $|x_1 u_1| > |x_2 u_2| \geq 0$, $|v_2 y_2| > |v_1 y_1| \geq 0$ and $\delta(q_0, u_1 \# v_1) = p$, $\delta(p, \$) = q$, $\delta(q, u_2 \# v_2) \in F$. If $|v_1 y_1| \geq 1$ and $|x_2 u_2| \geq 1$, then there are $x' = x_1 u_1 p \iota^{|v_1 y_1| - 1} \in L'$, $y' = \iota^{|x_2 u_2| - 1} q v_2 y_2 \in L'$ and $r' = (\lambda, p, q, \lambda) \in R'$ with $(x', y') \vdash_{r'} x_1 u_1 v_2 y_2$, which is in length-increasing mode since $|x'| = |x|$ and $|y'| = |y|$.

However, there are three cases in which z cannot be created by a length-increasing splicing in h':

(1) Suppose that $|v_1 y_1| = 0$ and $|x_2 u_2| \geq 1$. Then there are strings $x_1 u_1 p$ and $\iota^{|x_2 u_2| - 1} q v_2 y_2$ in L' and a rule $r' = (\lambda, p, q, \lambda) \in R'$ such that $(x_1 u_1 \mid p\,,\ \iota^{|x_2 u_2| - 1} q \mid v_2 y_2) \vdash_{r'} x_1 u_1 v_2 y_2$, which is in length-increasing mode if $|x_1 u_1| > |x_2 u_2|$, which follows from the original splicing $(x, y) \vdash_r^{in} z$, and $|v_2 y_2| > |p|$, which is only the case when in the original splicing indeed $|v_2 y_2| > 1$. This means that original splicings with $|v_1 y_1| = 0$ and $|v_2 y_2| = 1$, i.e., $(x_1 u_1 \mid, x_2 u_2 \mid a) \vdash^{in} x_1 u_1 a$ for an $a \in V$, cannot be simulated in h' (recall that $|v_2 y_2| = 0$ cannot occur when splicing in length-increasing mode).

(2) Suppose that $|x_2 u_2| = 0$ and $|v_1 y_1| \geq 1$. Following a similar reasoning it can be shown that original splicings with $|x_1 u_1| = 1$, i.e., $(a \mid v_1 y_1\,,\ \mid v_2 y_2) \vdash^{in} a v_2 y_2$ for an $a \in V$, cannot be simulated in h'.

(3) Suppose that $|v_1 y_1| = |x_2 u_2| = 0$. Then an original splicing $(x_1 u_1 \mid, \mid v_2 y_2) \vdash_r^{in} x_1 u_1 v_2 y_2$ can only be translated to $(x_1 u_1 \mid p\,,\ q \mid v_2 y_2) \vdash_{r'}^{in} x_1 u_1 v_2 y_2$ if $|x_1 u_1| > 1$ and $|v_2 y_2| > 1$. Hence we cannot simulate original splicings with $|x_1 u_1| = 1$ or $|v_2 y_2| = 1$.

Second attempt. In order to accommodate almost all these cases we add additional strings to the initial language L' and corresponding new rules to R'. Define $h'' = (V'', L'', R'')$ as

$$\begin{array}{rcll}
V'' & = & V \cup Q \cup \{\langle a{-}p\rangle, \langle a{+}p\rangle, \langle q{-}a\rangle, \langle q{+}a\rangle \mid a \in V,\ p,q \in Q\} & \\
L'' & = & L' \cup & \\
 & & \{w\langle a{-}p\rangle & \mid wa = xu \in L,\ a \in V \text{ and } \delta(q_0, u\#) = p\} \ \cup \\
 & & \{\iota^{|xu|-2}\langle q{+}a\rangle ab & \mid xub \in L,\ |xu| \geq 2,\ a,b \in V \text{ and} \\
 & & & \quad \delta(q, u\#) \in F \text{ or } \delta(q, u\#b) \in F\ \} \ \cup \\
 & & \{ba\langle a{+}p\rangle\iota^{|vy|-2} & \mid bvy \in L,\ |vy| \geq 2,\ a,b \in V \text{ and} \\
 & & & \quad \delta(q_0, \#v) = p \text{ or } \delta(q_0, b\#v) = p\} \ \cup \\
 & & \{\langle q{-}a\rangle w & \mid aw = vy \in L,\ a \in V \text{ and } \delta(q, \#v) \in F\ \} \\
R'' & = & R' \cup & \\
 & & \{(\lambda, \langle a{-}p\rangle, \langle q{+}a\rangle, \lambda), & \\
 & & \ (\lambda, \langle a{+}p\rangle, \langle q{-}a\rangle, \lambda) \ \mid \ \delta(p, \$) = q \text{ and } a \in V\} &
\end{array}$$

Intuitively, the symbol $\langle a - p\rangle$ signals that a symbol a was removed to code state p, whereas $\langle q{+}a\rangle$ indicates state q and the addition of symbol a.

Again, L'' can be constructed from L by a non-deterministic non-erasing GSM mapping, and R'' is finite.

The new strings and new rules can only splice among themselves, and simulate most of the remaining splicings of the original system (with regular rule set): let us reconsider the three cases that did not work in h'.

(1) Translating the original splicing $(x_1u_1 \mid,\ x_2u_2 \mid a) \vdash_r^{in} x_1u_1a$ now gives $(w \mid \langle b{-}p\rangle\ ,\ \iota^{|x_2u_2|-2}\langle q{+}b\rangle \mid ba) \vdash wba = x_1u_1a$, which is in length-increasing mode if and only if $|w| > |x_2u_2| - 1$. This means that it should be that $|x_1u_1| > |x_2u_2|$ and that follows from the original splicing. Hence all problems in case (1) are solved.

(2) Analogously, all problems in case (2) are solved.

(3) Translating the original splicing $(a \mid,\ \mid v_2y_2) \vdash_r^{in} av_2y_2$, for an $a \in V$, gives now $(ab \mid \langle b{+}p\rangle\ ,\ \langle q{-}b\rangle \mid w) \vdash abw = av_2y_2$, which is in length-increasing mode if and only if $|w| > 1$, i.e., $|v_2y_2| > 2$. Similarly, the other 'problem splicing', $(x_1u_1 \mid,\ \mid a) \vdash x_1u_1a$ can only be translated to h'' if $|x_1u_1| > 2$.

Consequently the only original splicings that cannot be simulated by h'' have $|x_2u_2| = |v_1y_1| = 0$ and $|x_1u_1| = 1$ while $1 \leq |v_2y_2| \leq 2$, or $|x_2u_2| = |v_1y_1| = 0$ and $|v_2y_2| = 1$ while $1 \leq |x_1u_1| \leq 2$. All these cases yield words of length two or three. $\square$

5.6 Summary

We have investigated the transition from regular rule sets to finite rule sets. As a result of that investigation we have given direct proofs for five equalities that are implicit in Table 3.1, i.e., $S(\mathcal{F}, \mathsf{FIN}) = S(\mathcal{F}, \mathsf{REG})$ for $\mathcal{F} \neq \mathsf{LIN}$, and for one new equality: $S(\mathsf{LIN}, \mathsf{FIN}) = S(\mathsf{LIN}, \mathsf{REG})$. Moreover, we have given a characterization of $S(\mathsf{LIN}, \mathsf{FIN})$ and $S(\mathsf{LIN}, \mathsf{REG})$, for which no exact position in the Chomsky hierarchy was known yet: $S(\mathsf{LIN}, \mathsf{FIN}) = S(\mathsf{LIN}, \mathsf{REG}) = \mathsf{LIN} \oplus \mathsf{LIN}$.

In the cases of same-length and length-decreasing splicing we could also prove that regular rule sets may be replaced by finite ones, while for self splicing we have shown that this is not even the case when starting with a regular initial language: $S_{sf}(\mathsf{REG}, \mathsf{FIN}) \subset S_{sf}(\mathsf{REG}, \mathsf{REG})$. For length-increasing splicing we proved that for each system h with a regular rule set there exists a system h' with a finite rule set such that $\sigma(h')$ consists of all words of $\sigma(h)$ that have length at least four. This means that $S_{in}(\mathcal{F}, \mathsf{FIN}) = S_{in}(\mathcal{F}, \mathsf{REG})$ provided that we do not consider words shorter than four symbols.

Since the families given in Table 3.3 are upper bounds and not (necessarily) characterizations, the above mentioned (in)equalities for restricted splicing are all new.

Chapter 6

Upper bounds for restricted non-iterated splicing

As we have discussed in Section 3.3, for restricted non-iterated splicing several families $S_\mu(\mathcal{F}_1, \mathcal{F}_2)$ still have unknown smallest upper bounds within the Chomsky hierarchy. We determine these upper bounds for non-iterated splicing in length-increasing, length-decreasing, same-length and self splicing mode, and we improve some of the known upper bounds.

The open problems indicated in Table 3.3 involve either a regular initial language and context-free splicing rules, or vice versa. For unrestricted non-iterated splicing the upper bounds for these two cases are determined in Lemma 3.3 and Lemma 3.6 of [HPP97]. We use the ideas from the proofs of these two lemma's to define, for each splicing system $h = (V, L, R)$, the language $C(L, R)$ that combines the initial language with the rules:

$$C(L,R) = \{\ x_1u_1\#v_1y_1\$x_2u_2\#v_2y_2 \ \mid\ x_1u_1v_1y_1\ ,\ x_2u_2v_2y_2 \in L \text{ and } u_1\#v_1\$u_2\#v_2 \in \mathcal{Z}(R)\}.$$

This language turns out to be very helpful in determining upper bounds for restricted splicing families. Note that $\sigma(h) = g(C(L, R))$, where g is the GSM mapping that erases the two #'s and everything in between from each word in $C(L, R)$.

Consider the two cases mentioned above, i.e., let $L \in \mathcal{F}_1$ and $\mathcal{Z}(R) \in \mathcal{F}_2$, with $\{\mathcal{F}_1, \mathcal{F}_2\} = \{\mathsf{REG}, \mathsf{CF}\}$. Using substitution of \$ with the regular set $V^*\$V^*$ and concatenation on both sides with the regular set V^*, the language $\mathcal{Z}(R)$ is transformed into the language $R' = \{x_1u_1\#v_1y_1\$x_2u_2\#v_2y_2 \mid u_1\#v_1\$u_2\#v_2 \in \mathcal{Z}(R) \text{ and } x_1, y_1, x_2, y_2 \in V^*\}$. Furthermore, using shuffle with symbols and concatenation, the language $L' = \{x\#y\$w\#z \mid xy, wz \in L\}$ can be constructed from L. Since both REG and CF are closed under these four operations, we either have $L' \in \mathsf{REG}$ and $R' \in \mathsf{CF}$, or $L' \in \mathsf{CF}$ and $R' \in \mathsf{REG}$. Clearly $C(L, R) = L' \cap R'$, which is a context-free language since CF is closed under

intersection with regular sets. This argument shows that both $S(\mathsf{REG},\mathsf{CF})$ and $S(\mathsf{CF},\mathsf{REG})$ are subfamilies of CF (cf. Table 3.1). In this chapter we seek to extend this argumentation to restricted splicing.

6.1 Same-length splicing

As can be seen in Table 3.3, for $S_{sl}(\mathcal{F}_1,\mathcal{F}_2)$ with $\mathcal{F}_1 = \mathsf{LIN},\mathsf{CF}$ and $\mathcal{F}_2 = \mathsf{FIN},\mathsf{REG}$ it is only known that all four families contain a non-context-free language, as shown by the following example.

Example 6.1 Let $h = (\{a,b,c,d\},L,R)$ be the splicing system defined by

$$\begin{aligned} L &= \{a^n b^n d \mid n \geq 1\} \cup \{d\, b^n c^n \mid n \geq 1\} \in \mathsf{LIN} \\ Z(R) &= \{a \# b \$ d \# b\} \in \mathsf{FIN} \end{aligned}$$

The form of the rules causes the first term of each splicing to be of the form $a^k b^k d$, and the second term of the form $d\, b^j c^j$, for some $k,j \geq 1$. Moreover, if we consider same-length splicing, we should have $k = j$. Then $(a^k \mid b^k d\, ,\, d \mid b^k c^k) \vdash^{sl} a^k b^k c^k$, using the only splicing rule in R. Consequently

$$\sigma_{sl}(h) = \{a^n b^n c^n \mid n \geq 1\}$$

which is not a context-free language. □

It is an open problem whether the smallest upper bound of $S_{sl}(\mathcal{F}_1,\mathcal{F}_2)$ in the Chomsky hierarchy is CS or RE, for $\mathcal{F}_1$ and $\mathcal{F}_2$ as above, and for $\mathcal{F}_1 = \mathsf{REG}$ and $\mathcal{F}_2 = \mathsf{LIN},\mathsf{CF}$. We solve this by proving that all languages in $S_{sl}(\mathsf{REG},\mathsf{CF})$ and $S_{sl}(\mathsf{CF},\mathsf{REG})$ are context-free valence languages over $\mathbb{Z}^1$ and thus context-sensitive (see Section 2.5).

Theorem 6.1 *$S_{sl}(\mathsf{REG},\mathsf{CF}) \subseteq \mathsf{CF}(\mathbb{Z}^1)$ and $S_{sl}(\mathsf{CF},\mathsf{REG}) \subseteq \mathsf{CF}(\mathbb{Z}^1)$.*

Proof. Let $h = (V,L,R)$ be a splicing system of (REG, CF) type or (CF, REG) type. As described before, the language $C(L,R) = \{\, x_1u_1\#v_1y_1\$x_2u_2\#v_2y_2 \mid x_1u_1v_1y_1\, ,\, x_2u_2v_2y_2 \in L$ and $u_1\#v_1\$u_2\#v_2 \in Z(R)\}$ is context-free.

Now $\sigma_{sl}(h) = \tau_{sl}(C(L,R))$, for the $\mathbb{Z}^1$-transducer τ_{sl} that transforms the string $x_1u_1\#v_1y_1\$x_2u_2\#v_2y_2 \in C(L,R)$ into $x_1u_1v_2y_2$ while at the same time checking whether $|x_1u_1v_1y_1| = |x_2u_2v_2y_2|$: it adds 1 for each symbol of $x_1u_1v_1y_1$, it subtracts 1 for each symbol of $x_2u_2v_2y_2$, and it copies x_1u_1 and v_2y_2 to the output.

Since $\mathsf{CF} = \mathsf{CF}(\mathbb{Z}^0)$ and since (as shown in Section 2.5) applying a $\mathbb{Z}^\ell$-transduction to a $\mathsf{CF}(\mathbb{Z}^k)$ language gives a $\mathsf{CF}(\mathbb{Z}^{k+\ell})$ language, we have $\sigma_{sl}(h) = \tau_{sl}(C(L,R)) \in \mathsf{CF}(\mathbb{Z}^1)$. □

This result enables us to fill in six of the missing entries in Table 3.3:
$S_{sl}(\mathsf{REG}, \mathcal{F}_2) \subseteq \mathsf{CF}(\mathbb{Z}^1)$ for $\mathcal{F}_2 = \mathsf{LIN}, \mathsf{CF}$, and
$S_{sl}(\mathcal{F}_1, \mathcal{F}_2) \subseteq \mathsf{CF}(\mathbb{Z}^1)$ for $\mathcal{F}_1 = \mathsf{LIN}, \mathsf{CF}$ and $\mathcal{F}_2 = \mathsf{FIN}, \mathsf{REG}$.

6.2 Splicing in *in* or *de* mode

First, we give an affirmative answer to a question asked in [KPS96, p.238]: does $S_{in}(\mathsf{REG}, \mathsf{LIN})$ contain a non-context-free language? We do this by adapting an example given in the proof of Lemma 10 from that same paper.

Example 6.2 Let $h = (\{a, b, c\}, L, R)$ be the splicing system defined by

$$\begin{aligned} L &= c\, b^* a^* b^* c \;\in\; \mathsf{REG} \\ \mathcal{Z}(R) &= \{cb^m a^n \# b^n c \$ c \# b^m c \mid m, n \geq 0\} \;\in\; \mathsf{LIN} \end{aligned}$$

The only splicings possible using the rule $cb^m a^n \# b^n c \$ c \# b^m c$ are of the form $(cb^m a^n \mid b^n c\, ,\, c \mid b^m c) \vdash cb^m a^n b^m c$. If the splicing has to be done in length-increasing mode, then we must have $m + n + 1 > 1$ and $m + 1 > n + 1$ (recall that the requirements $|x_1 u_1 v_2 y_2| > |x_1 u_1 v_1 y_1|$ and $|x_1 u_1 v_2 y_2| > |x_2 u_2 v_2 y_2|$ can be simplified to $|v_2 y_2| > |v_1 y_1|$ and $|x_1 u_1| > |x_2 u_2|$, respectively), hence

$$\sigma_{in}(h) = \{cb^m a^n b^m c \mid m, n \geq 0 \text{ and } m > n\}$$

which is not a context-free language. □

Hence we have the following result.

Lemma 6.2 $S_{in}(\mathsf{REG}, \mathsf{LIN}) - \mathsf{CF} \neq \varnothing$

Similar to the case of same-length splicing, we prove that both $S_{in}(\mathsf{REG}, \mathsf{CF})$ and $S_{in}(\mathsf{CF}, \mathsf{REG})$ are subfamilies of $\mathsf{CF}(\mathbb{Z}^2)$. For $S_{in}(\mathsf{REG}, \mathsf{CF})$ this answers the question whether the smallest upper bound in the Chomsky hierarchy is CS or RE, whereas for $S_{in}(\mathsf{CF}, \mathsf{REG})$ this improves the known upper bound CS.

Theorem 6.3 $S_{in}(\mathsf{REG}, \mathsf{CF}) \subseteq \mathsf{CF}(\mathbb{Z}^2)$ *and* $S_{in}(\mathsf{CF}, \mathsf{REG}) \subseteq \mathsf{CF}(\mathbb{Z}^2)$.

Proof. Let $h = (V, L, R)$ be a splicing system of $(\mathsf{REG}, \mathsf{CF})$ type or of $(\mathsf{CF}, \mathsf{REG})$ type. Construct the context-free language $C(L, R)$ from L and R as before. Now a non-deterministic $\mathbb{Z}^2$-transducer τ_{in} can transform each word $x_1 u_1 \# v_1 y_1 \$ x_2 u_2 \# v_2 y_2 \in C(L, R)$ for which $|x_1 u_1| > |x_2 u_2|$ and $|v_2 y_2| > |v_1 y_1|$ to $x_1 u_1 v_2 y_2$. To compare the lengths of $x_1 u_1$ and $x_2 u_2$ it uses its 'first counter' as follows: for the first letter of $x_1 u_1$ (note that, because of the requirements for *in* splicing, $x_1 u_1$ has to be non-empty) it adds 0, for each other letter of $x_1 u_1$ it non-deterministically adds 0 or 1, hence for $x_1 u_1$ an amount $\alpha < |x_1 u_1|$

is added. For each letter of x_2u_2 it subtracts 1, hence an amount $\beta = -|x_2u_2|$ is added. At the end of the computation this component of the counter has to have the value 0, which means that $\alpha + \beta = 0$, i.e., $\alpha = |x_2u_2|$ and thus $|x_2u_2| < |x_1u_1|$. The lengths of v_1y_1 and v_2y_2 are compared in an analogous way, using the second counter.

Clearly then $\sigma_{in}(h) = \tau_{in}(C(L,R)) \in \mathsf{CF}(\mathbb{Z}^2)$. □

We can now fill in two entries of Table 3.3 and improve four of the upper bounds given there:
$S_{in}(\mathsf{REG}, \mathcal{F}_2) \subseteq \mathsf{CF}(\mathbb{Z}^2)$ for $\mathcal{F}_2 = \mathsf{LIN}, \mathsf{CF}$, and
$S_{in}(\mathcal{F}_1, \mathcal{F}_2) \subseteq \mathsf{CF}(\mathbb{Z}^2)$ for $\mathcal{F}_1 = \mathsf{LIN}, \mathsf{CF}$ and $\mathcal{F}_2 = \mathsf{FIN}, \mathsf{REG}$.

The problem whether length-decreasing splicing of a regular initial language and a linear set of rules can yield a non-context-free language is also open ([KPS96]), and again it can be solved by adapting an (other) example given in the proof of Lemma 10 in [KPS96].

Example 6.3 Replace the initial language of Example 6.2 by

$$L' \;=\; c\,b^*a^*b^*c \;\cup\; c^*b^*c \;\in\; \mathsf{REG}$$

and let $h' = (\{a,b,c\}, L', R)$ with R as in Example 6.2. Now the only possible length-decreasing splicings are $(cb^m a^n \mid b^n c\,,\; c^\ell c \mid b^m c) \vdash^{de} cb^m a^n b^m c$, where $1 + m + n < \ell + 1$ and $m + 1 < n + 1$, thus

$$\sigma_{de}(h') = \{cb^m a^n b^m c \mid m, n \geq 0 \text{ and } m < n\}$$

which is not in CF. □

Consequently we have the following result.

Lemma 6.4 $S_{de}(\mathsf{REG}, \mathsf{LIN}) - \mathsf{CF} \neq \varnothing$

Clearly the constructions for *in* splicing can be adapted for *de* splicing, which gives the following theorem.

Theorem 6.5 $S_{de}(\mathsf{REG}, \mathsf{CF}) \subseteq \mathsf{CF}(\mathbb{Z}^2)$ *and* $S_{de}(\mathsf{CF}, \mathsf{REG}) \subseteq \mathsf{CF}(\mathbb{Z}^2)$.

Again, this gives the more general results
$S_{de}(\mathsf{REG}, \mathcal{F}_2) \subseteq \mathsf{CF}(\mathbb{Z}^2)$ for $\mathcal{F}_2 = \mathsf{LIN}, \mathsf{CF}$ and
$S_{de}(\mathcal{F}_1, \mathcal{F}_2) \subseteq \mathsf{CF}(\mathbb{Z}^2)$ for $\mathcal{F}_1 = \mathsf{LIN}, \mathsf{CF}$ and $\mathcal{F}_2 = \mathsf{FIN}, \mathsf{REG}$.

6.3 Self splicing

Self splicing linear or context-free initial languages with finite or regular rules can give a language outside CF, but can it also yield a non-context-sensitive language? We prove that the answer to this question is negative. We start by illustrating that even the self splicing of a regular initial language with a finite set of rules can lead outside CF.

Example 6.4 (See also [PRS98, proof of Theorem 11.1].)
Let $h = (\{a, b, c, d\}, L, R)$ be the splicing system defined by

$$\begin{aligned} L &= a\,b^*c^*d \;\in\; \mathsf{REG} \\ R &= \{\#\,d\,\$\,a\,\#\} \;\in \mathsf{FIN} \end{aligned}$$

Then $(ab^mc^k \mid d, a \mid b^mc^kd) \vdash^{sf} ab^mc^kb^mc^kd$ and thus

$$\sigma_{sf}(h) = \{ab^mc^kb^mc^kd \mid m, k \geq 0\}$$

which is not context-free. □

Theorem 6.6 $S_{sf}(\mathsf{CF}, \mathsf{REG}) \subseteq \mathsf{CS}$

Proof. Let $h = (V, L, R)$ be a splicing system with $L \in \mathsf{CF}$ and $\mathcal{Z}(R) \in \mathsf{REG}$. We show how to obtain an LBA that can check whether a word on its input tape belongs to $\sigma_{sf}(h)$ or not. Since the languages accepted by LBA's are exactly the context-sensitive languages, this proves that $S_{sf}(\mathsf{CF}, \mathsf{REG}) \subseteq \mathsf{CS}$.

As explained in Section 5.3, $\sigma_{sf}(h)$ can be described as $L_{\mathrm{xz}} \cup L_{\mathrm{xyyz}}$, where

$$\begin{aligned} L_{\mathrm{xz}} &= \{xz \mid xyz \in L \text{ with } u_1 \in \mathrm{Suf}\,(x), v_1 \in \mathrm{Pref}\,(yz), \\ &\qquad u_2 \in \mathrm{Suf}\,(xy), v_2 \in \mathrm{Pref}\,(z) \text{ for a } (u_1, v_1, u_2, v_2) \in R\} \text{ and} \\ L_{\mathrm{xyyz}} &= \{xyyz \mid xyz \in L \text{ with } u_1 \in \mathrm{Suf}\,(xy), v_1 \in \mathrm{Pref}\,(z), \\ &\qquad u_2 \in \mathrm{Suf}\,(x), v_2 \in \mathrm{Pref}\,(yz) \text{ for a } (u_1, v_1, u_2, v_2) \in R\}. \end{aligned}$$

Let w be the word that is on the tape of the LBA at the beginning of its computation. The LBA starts by guessing that w belongs either to L_{xz} or to L_{xyyz}.

In the case of L_{xz}, let 1 and 2 be symbols not in V and let $L_1 = \{x1y2z \mid xyz \in L \text{ with } u_1 \in \mathrm{Suf}\,(x), v_1 \in \mathrm{Pref}\,(yz), u_2 \in \mathrm{Suf}\,(xy), v_2 \in \mathrm{Pref}\,(z) \text{ for a } (u_1, v_1, u_2, v_2) \in R\}$. Note that L_1 can be obtained from L by a GSM mapping similar to the GSM mapping described in the proof of Theorem 5.5 (but this one has to search for the two cutting points simultaneously, because the splicing sites u_1v_1 and u_2v_2 can overlap and the GSM is not allowed to go back on its input). Moreover, L_{xz} can be constructed from L_1, also by a GSM mapping. This means that there exists a context-free grammar G with $L(G) = L_{\mathrm{xz}}$. Now

the LBA can check whether $w \in L(G)$ by simulating computations of G on its 'second track'.

In the case of L_{xyyz}, the LBA guesses a non-empty subword y and checks whether $w = xyyz$ and whether the splicing sites u_1v_1 and u_2v_2 occur in the right places. If so, then it may mark (or erase) each letter in the second copy of y, and then check whether the rest of w, i.e., xyz, belongs to the context-free language L. □

Hence we have $S_{sf}(\mathcal{F}_1, \mathcal{F}_2) \subseteq \mathsf{CS}$ for $\mathcal{F}_1 = \mathsf{LIN}, \mathsf{CF}$ and $\mathcal{F}_2 = \mathsf{FIN}, \mathsf{REG}$. In fact, this upper bound can be improved to the family 2DGSM(CF), which is a strict subfamily of CS (see [HV02]).

Example 6.4 implies that also self splicings of (REG, LIN) type and (REG, CF) type can yield languages outside CF. We prove that they can even define non-context-sensitive languages, i.e., we prove that the smallest upper bound in the Chomsky hierarchy of $S_{sf}(\mathsf{REG}, \mathsf{CF})$ and $S_{sf}(\mathsf{REG}, \mathsf{LIN})$ is RE.

Lemma 6.7 *Let L_1, L_2 be linear languages over Σ and let 0 be a symbol not in Σ. Then $0L_1/L_2 \in S_{sf}(\mathsf{REG}, \mathsf{CF})$.*

Proof. Assume that L_1 and L_2 are linear languages with $L_1, L_2 \subseteq \Sigma^*$ for some alphabet Σ. Let $0, 1, 2 \notin \Sigma$ be new symbols, and define $h = (\Sigma \cup \{0, 1, 2\}, L, R)$ by

$$\begin{aligned} L &= 0\,\Sigma^*\,1\,\Sigma^*\,2 \in \mathsf{REG} \\ \angle(R) &= \{0\,u\,\#\,1\,v\,2\,\$\,1\,w\,2\,\# \mid uv \in L_1, w \in L_2\} \end{aligned}$$

Since LIN is closed under concatenation with symbols and under shuffle with strings (but not under concatenation) we have $\angle(R) \in \mathsf{LIN} \cdot \mathsf{LIN} \subseteq \mathsf{CF}$.

We start by proving that $\sigma_{sf}(h) \subseteq 0L_1/L_2$. Let $x \in L$ and $(x, x) \vdash_r z$ for an $r = (0u, 1v2, 1w2, \lambda) \in R$. Then because of the form of the axioms and the first splicing site we must have $x = 0u1v2$. Moreover it must hold that $1v2 = 1w2$, i.e., $v = w$, because we are considering self splicing. Clearly $(0u \mid 1v2\ ,\ 0u1v2 \mid\ ) \vdash_r^{sf} z = 0u \in 0L_1/L_2$, since $uv \in L_1$ and $v = w \in L_2$ by construction.

Now take $z \in L_1/L_2$, i.e., there is a $y \in L_2$ such that $zy \in L_1$. According to the definition of h there is a splicing rule $r = (0z, 1y2, 1y2, \lambda) \in R$ and an axiom $0z1y2$, and so $(0z \mid 1y2\ ,\ 0z1y2 \mid\ ) \vdash_r 0z \in \sigma_{sf}(h)$. □

Since every RE language can be written as the quotient of two linear languages ([LLR85, Proposition 13]), Lemma 6.7 implies the following.

Corollary 6.8 *Let K be a language over Σ and let 0 be a symbol not in Σ. If $K \in \mathsf{RE}$, then $0K \in S_{sf}(\mathsf{REG}, \mathsf{CF})$.*

Since CS is closed under quotient with symbols, $0K \in \mathsf{CS}$ would imply $K \in \mathsf{CS}$. Consequently $K \in \mathsf{RE} - \mathsf{CS}$ implies $0K \in \mathsf{RE} - \mathsf{CS}$, thus the smallest upper bound in the Chomsky hierarchy of $S_{sf}(\mathsf{REG}, \mathsf{CF})$ is RE, as formulated in the following theorem.

Theorem 6.9 $S_{sf}(\mathsf{REG}, \mathsf{CF})$ *contains languages from* $\mathsf{RE} - \mathsf{CS}$.

The same result holds for $S_{sf}(\mathsf{REG}, \mathsf{LIN})$, i.e., we can give a construction that uses a set of rules R with $\mathcal{Z}(R) \in \mathsf{LIN}$ instead of the set R from the proof of Lemma 6.7, for which $\mathcal{Z}(R) \in \mathsf{LIN} \cdot \mathsf{LIN}$. We do this by reconsidering the above mentioned proof that every recursively enumerable language is the quotient of two linear languages. In the following example we use the main idea of that proof: one step of a Turing machine can be captured by a linear grammar, provided that we represent one of the two configurations involved by its mirror image. This idea originates from [Har67], where it is shown that every recursively enumerable set is the quotient of two context-free languages.

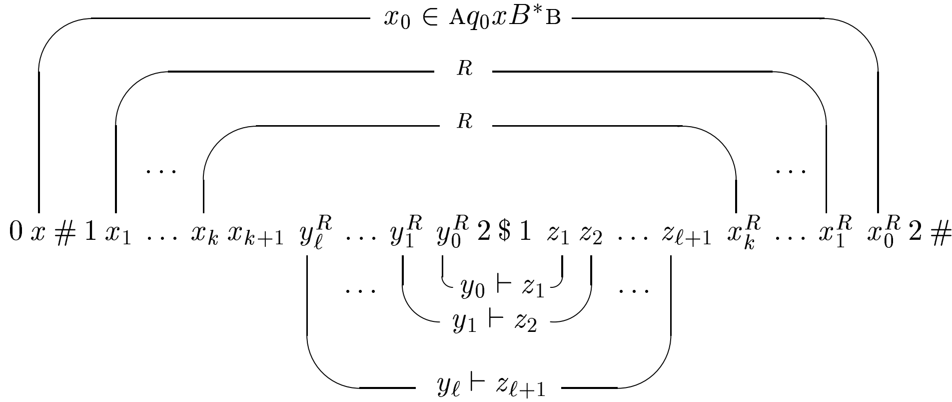


Figure 6.1: The structure of strings in $K_{\mathcal{M}}$ (Example 6.5)

Example 6.5 Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a Turing machine. For this example we write the configurations of $\mathcal{M}$ as strings of the form $\textsc{a}\Gamma^*Q\Gamma^*\textsc{b}$, for new symbols $\textsc{a}$ and $\textsc{b}$. We extend the notation $\vdash_{\mathcal{M}}$ for computational steps of $\mathcal{M}$ to this way of writing configurations.

Let the language $K_{\mathcal{M}}$ consist of the words

$$0\,x\,\#\,1\,x_1 \ldots x_k\,x_{k+1}\;y_\ell^R \ldots y_1^R\,y_0^R\,2\,\$\,1\,z_1\,z_2 \ldots z_{\ell+1}\;x_k^R \ldots x_1^R x_0^R\,2\,\#$$

where $0, 1, 2, \#, \$$ are new symbols,
for a string w, w^R denotes the mirror image of w,
$x \in \Sigma^*$,

$x_0, \ldots, x_{k+1}, y_0, \ldots, y_\ell, z_1, \ldots, z_{\ell+1} \in \text{A}\,\Gamma^*\,Q\,\Gamma^*\,\text{B}$, for $k, \ell \geq 0$,
$y_i \vdash_{\mathcal{M}} z_{i+1}$ for $0 \leq i \leq \ell$,
$x_0 \in \text{A}\,q_0\,x\,B^*\,\text{B}$, and
$x_{k+1} \in \text{A}\Gamma^*\,F\,\Gamma^*\text{B}$.

So each $x_i^R, x_{i+1}, y_j^R, z_{j+1}$, where $0 \leq i \leq k$ and $0 \leq j \leq \ell$, is (the mirror image of) a potential configuration of $\mathcal{M}$, and we use the new symbols A and B to separate them from each other. A picture may clarify the situation: see Figure 6.1.

$K_{\mathcal{M}}$ is a linear language, generated by the following grammar. The start symbol is S, a, b, c range over Γ, $d \in \Sigma$, $f \in F$ and $p, q \in Q$.

$S \to 0\,T\,2\,\#$	$U' \to a\,U'\,a \mid p\,U''\,p$
$T \to T'\,q_0\,\text{A}$	$U'' \to a\,U''\,a \mid \text{B}\,U\,\text{B}$
$T' \to d\,T'\,d \mid T''$	$V \to \text{A}\,V'$
$T'' \to T''\,B \mid \#\,1\,U\,\text{B}$	$V' \to a\,V' \mid f\,V''$
$U \to \text{A}\,U'\,\text{A} \mid V$	$V'' \to a\,V'' \mid \text{B}\,W$

$W \to \text{B}\,W'\,\text{B}$	
$W \to \text{B}\,a\,p\,W''\,b\,q\,B\,\text{B}$	if $(p, a, q, b, R) \in \delta$
$W' \to a\,W'\,a$	
$W' \to a\,p\,c\,W''\,q\,c\,b$	if $(p, a, q, b, L) \in \delta$
$W' \to a\,p\,\text{A}\,W\,\text{A}\,q\,B\,b \mid a\,p\,\text{A}\,2\,\$\,1\,\text{A}\,q\,B\,b$	if $(p, a, q, b, L) \in \delta$
$W' \to a\,p\,W''\,b\,q$	if $(p, a, q, b, R) \in \delta$
$W' \to a\,p\,W''\,q\,b$	if $(p, a, q, b, N) \in \delta$
$W'' \to a\,W''\,a \mid \text{A}\,W\,\text{A} \mid \text{A}\,2\,\$\,1\,\text{A}$	

Here T creates $x\,\#\,1$ and x_0^R, U generates x_i and x_i^R for each $i \in \{1, \ldots, k\}$, V derives x_{k+1} and W generates y_i^R and z_{i+1} for each $i \in \{0, \ldots, \ell\}$. Note that, since a, b and c may also stand for the blank symbol, we can get configurations with a lot of unnecessary blanks, but we do not miss any configuration. □

Theorem 6.10 *Let K be a language over Σ and let 0 be a symbol not in Σ. If $K \in \mathsf{RE}$, then $0K \in S_{sf}(\mathsf{REG}, \mathsf{LIN})$.*

Proof. *Construction.* Let $K = L(\mathcal{M})$ for a deterministic Turing machine $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, and let $K_{\mathcal{M}}$ be the linear language as defined in Example 6.5. Now $\sigma_{sf}(h) = 0K$, for the splicing system $h = (V, L, R)$ defined by

$$\begin{aligned} V &= \Gamma \cup \{0,1,2\} \cup Q \cup \{\text{A}, \text{B}\} \\ L &= 0\,\Sigma^*\,1\,(\Gamma \cup Q \cup \{\text{A}, \text{B}\})^*\,2 \in \mathsf{REG} \\ \angle(R) &= K_{\mathcal{M}} \in \mathsf{LIN} \end{aligned}$$

where $0, 1, 2 \notin \Gamma \cup Q \cup \{\text{A}, \text{B}\}$.

Correctness. The splicing rules of h are of the form

$$\frac{\quad 0\,x \mid 1\,x_1\dots x_{k+1}\,y_\ell^R\dots y_1^R y_0^R\,2}{1\,z_1\dots z_{\ell+1}\,x_k^R\dots x_1^R x_0^R\,2 \mid \quad}$$

with $x_1,\dots,x_{k+1},y_0,\dots,y_\ell,z_1,\dots,z_{\ell+1} \in \mathrm{A}\,\Gamma^*\,Q\,\Gamma^*\,\mathrm{B}$, $x_0 \in \mathrm{A}\,q_0\,x\,B^*\,\mathrm{B}$ and $y_i \vdash_{\mathcal{M}} z_{i+1}$ for $0 \le i \le \ell$.

Because of the form of the initial strings and of the rules, the first term of the splicing must be of the form $0x1x_1\dots x_{k+1}y_\ell^R\dots y_1^R y_0^R 2$. Since we consider self splicing, this is also the second term. The second splicing site now enforces the equality

$$x_1\dots x_{k+1}\,y_\ell^R\dots y_1^R y_0^R = z_1\dots z_{\ell+1}\,x_k^R\dots x_1^R x_0^R,$$

and the marking with A and B ensures that $k=\ell$, $x_i = z_i$ for $1 \le i \le k+1$ and $y_j = x_j$ for $0 \le j \le k$. Hence $x_0 \in \mathrm{A}\,q_0\,x\,B^*\,\mathrm{B}$ is the initial configuration of $\mathcal{M}$ for the input word x, $x_i = y_i \vdash_{\mathcal{M}} z_{i+1} = x_{i+1}$ for $0 \le i \le k$, and x_{k+1} is the end configuration of $\mathcal{M}$ for x. Thus $x_0 \vdash_{\mathcal{M}} x_1 \vdash_{\mathcal{M}} \dots \vdash_{\mathcal{M}} x_{k+1}$ is an accepting configuration sequence for x. Consequently, if $0\;x\;1\;x_1\dots x_{k+1}\;y_\ell^R\dots y_1^R y_0^R\;2$ splices with itself to give $0\,x$, then $x \in L(\mathcal{M})$.

For the proof in the reverse direction, the above can be read backwards. □

Theorem 6.11 *$S_{sf}(\mathsf{REG},\mathsf{LIN})$ contains languages from* $\mathsf{RE}-\mathsf{CS}$.

6.4 Summary

We have solved all open problems indicated in Table 3.3, and improved some of the known CS upper bounds given there. In the following table we summarize the results on the upper bounds of the four restricted splicing modes that we considered.

$\mathcal{F}_2 \rightarrow$	FIN	REG	LIN	CF	FIN	REG	LIN	CF
f	REG	REG	LIN	CF	CF	CF	RE	RE
in	REG	REG	CF($\mathbb{Z}^2$)	CF($\mathbb{Z}^2$)	CF($\mathbb{Z}^2$)	CF($\mathbb{Z}^2$)	CS	CS
de	REG	REG	CF($\mathbb{Z}^2$)	CF($\mathbb{Z}^2$)	CF($\mathbb{Z}^2$)	CF($\mathbb{Z}^2$)	RE	RE
sl	LIN	LIN	CF($\mathbb{Z}^1$)	CF($\mathbb{Z}^1$)	CF($\mathbb{Z}^1$)	CF($\mathbb{Z}^1$)	RE	RE
sf	CS	CS	RE	RE	CS	CS	RE	RE
	$\mathcal{F}_1 = \mathsf{REG}$				$\mathcal{F}_1 = \mathsf{LIN},\mathsf{CF}$			

Table 6.1: Upper bounds of $S_\mu(\mathcal{F}_1,\mathcal{F}_2)$ – updated

Part II

Sticker systems

Chapter 7

Definitions, examples and research topics

7.1 Sticker systems

Sticker systems are introduced in [KP+98] as a formal language model for the self-assembly phase of Adleman's experiment ([Adl94]). Self-assembly is the ability of complementary parts of single stranded pieces of DNA to stick together, thereby possibly leaving an overhanging 'sticky end' to which another part of single stranded DNA can stick, and so on. In this way a (partially) double stranded piece of DNA is created.

Fully double stranded DNA molecules can be written as a pair of 'matching' strings over the alphabet $\{\mathsf{a},\mathsf{c},\mathsf{g},\mathsf{t}\}$ of bases. Alternatively, for the matching base pairs we may use the symbols $\binom{\mathsf{a}}{\mathsf{t}}$, $\binom{\mathsf{t}}{\mathsf{a}}$, $\binom{\mathsf{c}}{\mathsf{g}}$, $\binom{\mathsf{g}}{\mathsf{c}}$. We appreciate both approaches, and will not distinguish between a (two-dimensional) pair of matching strands like $(\mathsf{agac},\mathsf{tctg})$ and a (one-dimensional) string of paired bases like $\binom{\mathsf{a}}{\mathsf{t}}\binom{\mathsf{g}}{\mathsf{c}}\binom{\mathsf{a}}{\mathsf{t}}\binom{\mathsf{c}}{\mathsf{g}}$.

For our purposes we will consider an alphabet Σ of 'abstract' DNA bases, and a relation $\rho \subseteq \Sigma \times \Sigma$ representing the complementarity relation. We extend ρ to a subset of $\Sigma^* \times \Sigma^*$ by demanding that two strings are complementary if they are of equal length and their letters are one by one complementary: $(a_1 \dots a_n, b_1 \dots b_m) \in \rho$, for $a_i, b_j \in \Sigma$, $1 \leq i \leq n$, $n \geq 0$, and $1 \leq j \leq m$, $m \geq 0$, if $n = m$ and $(a_\ell, b_\ell) \in \rho$ for each $1 \leq \ell \leq n$.

We define the alphabet Σ_ρ, representing matching pairs of symbols, as consisting of all symbols $\binom{a}{b}$ where $(a,b) \in \rho$ and $a, b \in \Sigma$. As explained above, we identify Σ_ρ^* with the subset ρ of $\Sigma^* \times \Sigma^*$: $\binom{a_1}{b_1} \dots \binom{a_n}{b_n}$ represents the same double stranded molecule as $(a_1 \dots a_n, b_1 \dots b_n)$, for $(a_i, b_i) \in \rho$ and $a_i, b_i \in \Sigma$. Note that thus $\lambda \in \Sigma_\rho^*$ equals $(\lambda, \lambda) \in \rho$. We will also use $\binom{a_1 \dots a_n}{b_1 \dots b_n}$ as an abbreviation of $\binom{a_1}{b_1} \dots \binom{a_n}{b_n}$.

We give a definition of sticker systems that is equivalent to the definition

given in [KP$^+$98], but stated directly in terms of strings, avoiding a lengthy definition of a 'sticker operation'. In [FP$^+$98], [PR98] and [PRS98] more general sticker systems are investigated; the sticker systems from [KP$^+$98], which we consider here, are called simple regular sticker systems in [PR98, PRS98].

Definition 7.1 A *sticker system* is a 5-tuple $\gamma = (\Sigma, \rho, D_u, D_\ell, A)$ where Σ is an alphabet, $\rho \subseteq \Sigma \times \Sigma$ is the complementarity relation, $D_u, D_\ell \subseteq \Sigma^+$ are finite sets of *upper* and *lower stickers*, and $A \subseteq \Sigma^* \times \Sigma^*$ is a finite set of *axioms.* □

We call a pair $(x_0x_1 \dots x_n, y_0y_1 \dots y_m) \in \rho$ a *(complete) computation of* γ if $(x_0, y_0) \in A$, $x_1, \dots, x_n \in D_u$ and $y_1, \dots, y_m \in D_\ell$, for some $n, m \geq 0$. If $n = m$, then $(x_0x_1 \dots x_n, y_0y_1 \dots y_m)$ is called a *fair* computation of γ. If there are no i and j, with $0 \leq i < n$ and $0 \leq j < m$, such that $(x_0x_1 \dots x_i, y_0y_1 \dots y_j)$ is a complete computation of γ, then $(x_0x_1 \dots x_n, y_0y_1 \dots y_m)$ is called a *primitive* computation. A computation that is both primitive and fair is called *primitive fair.* If we do not care about the exact composition of a computation we write '$(x, y) \in \rho$ is a computation', meaning that there are $(x_0, y_0) \in A$, $x_1, \dots, x_n \in D_u$ and $y_1, \dots, y_m \in D_\ell$ such that $x = x_0x_1 \dots x_n$ and $y = y_0y_1 \dots y_m$. We say that $(x_0x_1 \dots x_i, y_0y_1 \dots y_j)$, for some $i, j \geq 0$, is a *partial computation* of γ if $(x_0, y_0) \in A$, $x_1, \dots, x_i \in D_u$, $y_1, \dots, y_j \in D_\ell$, and if $|x_0x_1 \dots x_i| \leq |y_0y_1 \dots y_j|$ implies that there is a $w \in \text{Pref}\,(y_0y_1 \dots y_j)$ such that $(x_0x_1 \dots x_i, w) \in \rho$, whereas $|x_0x_1 \dots x_i| > |y_0y_1 \dots y_j|$ implies that there is a $v \in \text{Pref}\,(x_0x_1 \dots x_i)$ such that $(v, y_0y_1 \dots y_j) \in \rho$.

Note that in a primitive computation only at the end a so-called 'blunt end' occurs – i.e., the end of the computation is the first position where there is no sticky end – whereas in a non-primitive computation there is additionally at least one 'intermediate blunt end'.

Although we are mainly interested in the formal language theoretic aspects of sticker systems, we keep the analogy with molecules and DNA in mind, and will often write, for instance, '$(x, y) \in \rho$ is a double stranded string' or 'x is the upper strand of $(x, y) \in \rho$'.

Observe that our definition of computation differs from the one used in the literature: there a computation is a sequence of what we call partial computations, where the first element of the sequence is an axiom, the next is an extension of the previous element with one sticker, and so on until the last element, which is completely double stranded. In other words, in the literature a computation is defined both by the axiom and stickers used in it, and by the order in which the stickers are added. Hence one computation $(x_0x_1 \dots x_n, y_0y_1 \dots y_m)$ in our terminology corresponds to several computations according to the definition from the literature.

In our definition we have abstracted from the order in which stickers are added to get a complete computation. Indeed, this order has no effect on the result being a complete computation nor on the fairness or primitivity of

the computation. Nevertheless, we will sometimes find it useful to think of a computation as if it was constructed by adding stickers in a certain order.

7.2 Sticker languages

Let $\gamma = (\Sigma, \rho, D_u, D_\ell, A)$ be a sticker system. The (unrestricted) *molecular language* generated by γ is defined as

$$ML(\gamma) = \{ (x,y) \in \rho \mid (x,y) \text{ is a computation of } \gamma \}.$$

The *fair* molecular language generated by γ is defined as

$$ML_f(\gamma) = \{ (x,y) \in \rho \mid (x,y) \text{ is a fair computation of } \gamma \},$$

the *primitive* molecular language as

$$ML_p(\gamma) = \{ (x,y) \in \rho \mid (x,y) \text{ is a primitive computation of } \gamma \}$$

and the *primitive fair* molecular language as

$$ML_{pf}(\gamma) = \{ (x,y) \in \rho \mid (x,y) \text{ is a primitive fair computation of } \gamma \}$$

Furthermore, the (unrestricted) *sticker language* generated by γ is the projection onto the first (upper) component of the molecular language,

$$L(\gamma) = \{ x \in \Sigma^* \mid (x,y) \in ML(\gamma) \text{ for some } y \in \Sigma^* \}$$

and analogously for $L_f(\gamma)$, $L_p(\gamma)$ and $L_{pf}(\gamma)$, the fair, primitive and primitive fair sticker languages generated by γ, respectively.

Example 7.1 Let $\gamma = (\Sigma, \rho, D_u, D_\ell, A)$ be the sticker system defined by

$$\begin{aligned} \Sigma &= \{a, b\} \\ \rho &= id \\ D_u &= \{aa, ab\} \\ D_\ell &= \{aa, b\} \\ A &= \{ (ba, b) \, , \, (b, ba) \, , \, (bb, bb) \} \end{aligned}$$

Here *id* denotes the identity relation. Four sample complete computations of γ, given by their upper and lower strand, are depicted below. We have indicated the beginning and end of axioms with $[$ and $\rangle$, respectively, and the beginning and end of each sticker with $\langle$ and $\rangle$, respectively.

$[b\rangle\langle a\ b\rangle$	$[b\ a\rangle\langle a\ b\rangle$	$[b\rangle\langle aa\rangle\langle a\ b\rangle\langle aa\rangle$	$[b\ a\rangle\langle aa\rangle\langle a\ b\rangle\langle aa\rangle\langle aa\rangle$
$[b\ a\rangle\langle b\rangle$	$[b\rangle\langle aa\rangle\langle b\rangle$	$[b\ a\rangle\langle aa\rangle\langle b\rangle\langle aa\rangle$	$[b\rangle\langle aa\rangle\langle aa\rangle\langle b\rangle\langle aa\rangle\langle aa\rangle$

The first of these computations is both fair and primitive, the second is primitive but not fair, the third is fair but not primitive, and the last one is neither primitive nor fair. It is fairly easy to argue that

$$\begin{aligned} L(\gamma) &= ba^*b(aa)^* \\ L_p(\gamma) &= ba^*b \\ L_f(\gamma) &= b(aa)^*ab(aa)^* \ \cup \ bb(aa)^* \\ L_{pf}(\gamma) &= b(aa)^*ab \ \cup \ bb \end{aligned}$$

□

Example 7.2 Consider the following sticker system, a slight extension of the one given in the proof of Theorem 3 in [KP$^+$98]:
$\gamma = (\{a,b,c\}, \rho, D_u, D_\ell, A)$ with

$$\begin{aligned} \rho &= \{(a,a),(b,b),(b,c)\} \\ D_u &= \{aa, b\} \\ D_\ell &= \{a, bc\} \\ A &= \{(\lambda,\lambda)\} \end{aligned}$$

All computations of γ are composed of 'blocks' that consist either of aa in the upper strand and aa in the lower strand, or of bb in the upper strand and bc in the lower strand. Moreover, each of the blocks containing only a's uses one upper sticker and two lower stickers, whereas each other block uses two upper stickers and one lower sticker, as shown below.

$$\begin{array}{l} [\rangle\ \langle a \quad a\rangle\ \langle b\rangle\langle b\rangle\ \langle b\rangle\langle b\rangle\ \langle a \quad a\rangle \\ [\rangle\ \langle a\rangle\langle a\rangle\ \langle b \quad c\rangle\ \langle b \quad c\rangle\ \langle a\rangle\langle a\rangle \end{array}$$

Hence only one computation of γ is primitive: the one that uses only the axiom. Furthermore, the only way to make a computation fair is to ensure that it uses an equal amount of both kinds of blocks. Now clearly

$$\begin{aligned} ML(\gamma) &= \{ \ \tbinom{aa}{aa}, \tbinom{bb}{bc} \ \}^* \\ ML_p(\gamma) &= \{ \ \lambda \ \} \\ ML_f(\gamma) &= \{ \ \tbinom{x}{y} \in ML(\gamma) \mid \#_a(x) = \#_b(x) \ \} \\ ML_{pf}(\gamma) &= \{ \ \lambda \ \} \end{aligned}$$

and consequently

$$\begin{aligned} L(\gamma) &= \{aa, bb\}^* \\ L_f(\gamma) &= \{x \in L(\gamma) \mid \#_a(x) = \#_b(x)\} \end{aligned}$$

□

Obviously, for each sticker system γ, $L(\gamma)$ can be obtained from $ML(\gamma)$ by applying a coding. However, we may not reverse this: in general $ML(\gamma)$ is not the image of $L(\gamma)$ under an inverse coding, as is clear from Example 7.2.

In this thesis we consider sticker languages rather than the molecular variants. This can be justified using the following representation result, that implies that a sticker system γ can always be changed into a sticker system γ' such that the *sticker* language of γ' gives exactly the same information as the *molecular* language of γ.

Theorem 7.1 *For every sticker system $\gamma = (\Sigma, \rho, D_u, D_\ell, A)$ there is a sticker system $\gamma' = (\Sigma_\rho, id, D'_u, D'_\ell, A')$ such that $ML(\gamma) = L(\gamma')$.*

Proof. Let $h_u : \Sigma_\rho^* \to \Sigma^*$ be the homomorphism that maps $\binom{a}{b}$ to a, and let $h_\ell : \Sigma_\rho^* \to \Sigma^*$ be the homomorphism that maps $\binom{a}{b}$ to b. Note that $(x, y) \in \rho$ if and only if $\binom{x}{y} \in h_u^{-1}(x) \cap h_\ell^{-1}(y)$. In words, given $(x, y) \in \rho$, if we guess a 'lower strand' y' complementary to x (using h_u^{-1}) and an 'upper strand' x' complementary to y (using h_ℓ^{-1}), then we have guessed right if and only if $\binom{x}{y'} = \binom{x'}{y}$.

Construction. We construct $\gamma' = (\Sigma_\rho, id, D'_u, D'_\ell, A')$ as follows:

$$\begin{aligned} D'_u &= h_u^{-1}(D_u) \\ D'_\ell &= h_\ell^{-1}(D_\ell) \\ A' &= \{(x, y) \mid (h_u(x), h_\ell(y)) \in A\} \end{aligned}$$

Correctness. We start by proving that $ML(\gamma) \subseteq L(\gamma')$. Let $(x, y) \in ML(\gamma)$, i.e., $(x, y) \in \rho$ and there are $(x_0, y_0) \in A$, $x_1, \ldots, x_n$ in D_u and $y_1, \ldots, y_m$ in D_ℓ, with $n, m \geq 0$, such that $x = x_0 x_1 \ldots x_n$ and $y = y_0 y_1 \ldots y_m$. Then x and y can also be written as follows:

$$\begin{aligned} x &= x_0\, x_1 \ldots x_n &= x'_0\, x'_1 \ldots x'_m \\ y &= y'_0\, y'_1 \ldots y'_n &= y_0\, y_1 \ldots y_m \end{aligned}$$

where x_i, y_j as before, $(x_i, y'_i) \in \rho$ and $(x'_j, y_j) \in \rho$ (for $0 \leq i \leq n$ and $0 \leq j \leq m$). According to the definitions of A', D'_u and D'_ℓ, this means that $(\binom{x_0}{y'_0}, \binom{x'_0}{y_0}) \in A'$, $\binom{x_i}{y'_i} \in D'_u$ and $\binom{x'_j}{y_j} \in D'_\ell$ for $1 \leq i \leq n$ and $1 \leq j \leq m$. This implies that $(\binom{x_0}{y'_0}\binom{x_1}{y'_1} \ldots \binom{x_n}{y'_n}, \binom{x'_0}{y_0}\binom{x'_1}{y_1} \ldots \binom{x'_m}{y_m}) = (\binom{x}{y}, \binom{x}{y}) \in ML(\gamma')$, hence $\binom{x}{y} = (x, y) \in L(\gamma')$.

Note that the reasoning above is also valid if read backwards, which proves $L(\gamma') \subseteq ML(\gamma)$, and consequently $L(\gamma') = ML(\gamma)$. □

Since the stickers (axioms) of γ' have the same length as the corresponding stickers (axioms) of γ, the number of upper and lower stickers used in a computation does not change when passing from γ to γ'. Consequently both fairness and primitivity are preserved.

Corollary 7.2 *For every sticker system* $\gamma = (\Sigma, \rho, D_u, D_\ell, A)$ *there is a sticker system* $\gamma' = (\Sigma_\rho, id, D'_u, D'_\ell, A')$ *such that* $ML_f(\gamma) = L_f(\gamma')$, $ML_p(\gamma) = L_p(\gamma')$ *and* $ML_{pf}(\gamma) = L_{pf}(\gamma')$.

From Theorem 7.1 we can derive a rather unexpected but useful normal form for sticker systems: without changing the sticker language, we can always replace the complementarity relation ρ by the identity id on the alphabet Σ. Note that, of course, the molecular language *does* change if ρ was not already equal to id.

The idea behind this derivation is the following: instead of guessing double stranded lower stickers for the new system starting from lower stickers in the original system, we can guess just the upper strands corresponding to the original lower stickers (and similar for the axioms; the upper stickers remain the same). In other words, we use the same procedure as in the proof of Theorem 7.1, but we use only part of the answers that result from it.

Theorem 7.3 *For every sticker system* $\gamma = (\Sigma, \rho, D_u, D_\ell, A)$ *a sticker system* $\gamma' = (\Sigma, id, D_u, D'_\ell, A')$ *can be constructed with* $L(\gamma') = L(\gamma)$.

Proof. *Construction.* Let D'_ℓ and A' be defined as

$$\begin{aligned} D'_\ell &= \{w \in \Sigma^+ \mid (w, v) \in \rho \text{ for some } v \in D_\ell\} \\ A' &= \{(x_0, z_0) \mid (x_0, y_0) \in A \text{ for some } y_0, \text{ and } (z_0, y_0) \in \rho\} \end{aligned}$$

Correctness. Assume that $x \in L(\gamma)$, i.e., there is a y such that $(x, y) \in \rho$, $x = x_0 x_1 \dots x_n$ and $y = y_0 y_1 \dots y_m$, where $(x_0, y_0) \in A$, $x_i \in D_u$ for $1 \le i \le n$ and $n \ge 0$, and $y_j \in D_\ell$ for $1 \le j \le m$ and $m \ge 0$. Then x can also be written as $x = z_0 z_1 \dots z_m$, where $(z_k, y_k) \in \rho$ for $0 \le k \le m$. According to the definition of A' then $(x_0, z_0) \in A'$, and from the definition of D'_ℓ it follows that $z_j \in D'_\ell$ for $1 \le j \le m$. Therefore $(x_0 x_1 \dots x_n, z_0 z_1 \dots z_m) = (x, x) \in ML(\gamma')$, hence $x \in L(\gamma')$.

To prove that $L(\gamma') \subseteq L(\gamma)$, the above can be read backwards. $\square$

Obviously, again the number of stickers used and the lengths of the stickers are not changed, hence the following holds as well.

Corollary 7.4 *For every sticker system* $\gamma = (\Sigma, \rho, D_u, D_\ell, A)$ *a sticker system* $\gamma' = (\Sigma, id, D_u, D'_\ell, A')$ *can be constructed with* $L_f(\gamma') = L_f(\gamma)$, $L_p(\gamma') = L_p(\gamma)$ *and* $L_{pf}(\gamma') = L_{pf}(\gamma)$.

7.3 Families of sticker languages

The family of all sticker languages is denoted SL, while the families of fair, primitive and primitive fair sticker languages are denoted SL_f, SL_p and SL_{pf}, respectively.

We recall from the literature the results concerning the relations between SL, SL_f, SL_p, SL_{pf} and the Chomsky families. We do this rather elaborately, because we will use some of these constructions later on.

First, $\mathsf{SL} \subseteq \mathsf{REG}$. When a sticker is added to a partial computation, this can always be done in the strand opposite to the strand containing the current sticky end. Therefore, the current sticky end never needs to be longer than the maximal lengths of the stickers and the two components of each axiom. Consequently, a finite number of states of a finite automaton, where each state stands for a specific sticky end, can control the computation in the same way as the sticky ends do ([KP$^+$98, Lemma 1]).

Second, discarding in the previous construction the transitions that allow a complete computation to be continued yields a finite automaton for the primitive sticker language of the sticker system under consideration ([KP$^+$98, Lemma 2]), hence $\mathsf{SL}_p \subseteq \mathsf{REG}$. Indeed, primitivity is a 'local' property, hence easy to check using the states of a finite automaton.

Example 7.3 Consider the sticker system γ given in Example 7.1, that has upper stickers aa and ab, lower stickers aa and b, and axioms $(ba, b), (b, ba)$ and (bb, bb). From the definition of γ we derive that, when constructing sticker by sticker a complete computation of γ, we will never need other sticky ends than the following: one a in the upper or lower strand (written as $\genfrac{}{}{0pt}{}{a}{\lambda}$ or $\genfrac{}{}{0pt}{}{\lambda}{a}$, respectively), one b in the upper strand ($\genfrac{}{}{0pt}{}{b}{\lambda}$), the blunt end ($\genfrac{}{}{0pt}{}{\lambda}{\lambda}$) and aa in the upper or lower strand ($\genfrac{}{}{0pt}{}{aa}{\lambda}$ or $\genfrac{}{}{0pt}{}{\lambda}{aa}$).

As an example, observe the following sequence of seven (partial) computations of γ.

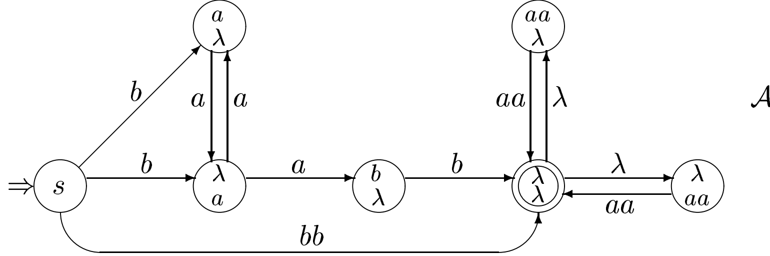
$$\begin{array}{l}[b\rangle\\ [b\ a\rangle\end{array}\quad \begin{array}{l}[b\rangle\langle aa\rangle\\ [b\ a\rangle\end{array}\quad \begin{array}{l}[b\rangle\langle aa\rangle\\ [b\ a\rangle\langle aa\rangle\end{array}\quad \begin{array}{l}[b\rangle\langle aa\rangle\langle ab\rangle\\ [b\ a\rangle\langle aa\rangle\end{array}\quad \begin{array}{l}[b\rangle\langle aa\rangle\langle a\ b\rangle\\ [b\ a\rangle\langle aa\rangle\langle b\rangle\end{array}$$

$$\begin{array}{l}[b\rangle\langle aa\rangle\langle a\ b\rangle\\ [b\ a\rangle\langle aa\rangle\langle b\rangle\langle aa\rangle\end{array}\qquad \begin{array}{l}[b\rangle\langle aa\rangle\langle a\ b\rangle\langle aa\rangle\\ [b\ a\rangle\langle aa\rangle\langle b\rangle\langle aa\rangle\end{array}$$

We use this sequence to explain how to construct a lazy finite automaton $\mathcal{A}$ such that $L(\mathcal{A}) = L(\gamma)$. Its states represent the occurring sticky ends, while its transitions are labelled with the prefix of the applied sticker that matches the current sticky end. We add a new initial state s from which the sticky ends resulting from the axioms are reached.

The sequence starts with the axiom (b, ba), meaning that the first letter of the string is a b and we have a sticky end $\genfrac{}{}{0pt}{}{\lambda}{a}$; in the automaton this is described

by the transition $(s, b, \binom{\lambda}{a})$. Then an upper sticker aa is used, adding the letter a to the string and leaving an overhang a in the upper strand; for this we add the transition $(\binom{\lambda}{a}, a, \binom{a}{\lambda})$, and so on.



Clearly, $L(\mathcal{A}) = ba^*b(aa)^* = L(\gamma)$. When considering only the primitive computations of γ it suffices to remove all paths that continue from the final state $\binom{\lambda}{\lambda}$ of $\mathcal{A}$, since that is the state that marks an intermediate blunt end. $\square$

Third, not every regular language can be generated as the unrestricted language of a sticker system: $\mathsf{REG} \not\subseteq \mathsf{SL}$. Suppose that γ is a sticker system that generates ba^*b. Since the number of a's in words from ba^*b can be arbitrarily large, there must be an upper sticker a^ℓ and a lower sticker a^k in γ, with $\ell, k \geq 1$. But then a complete computation for ba^ib, for some $i \geq 0$, can always be extended to a (non-primitive) complete computation for $ba^iba^{\ell k}$ ([PR98, Theorem 10]).

Fourth, we have $\mathsf{REG} \subseteq \mathsf{COD(SL)}$, which can be seen as follows. Sticker systems are formalizations of Adleman's use of self-assembly to find paths in a given graph. Since a finite automaton is also a graph, for each finite automaton a sticker system can be constructed of which the computations represent paths from the intial state to a final state of the automaton. We need a coding to remove the symbols representing the states from the resulting strings ([PRS98, Theorem 4.8], [KP$^+$98, Lemma 5]).

Fifth, $\mathsf{REG} \subseteq \mathsf{COD(SL}_p)$: the stickers in the above mentioned system are defined in such a way that the first symbol of an upper sticker never matches the first symbol of a lower sticker, thereby ensuring that each computation of the finite automaton is simulated by a primitive computation of the corresponding sticker system ([PRS98, Theorem 4.8]).

Sixth, addition of a few extra axioms to the above construction yields a fair computation for each computation of the finite automaton. Consequently, $\mathsf{REG} \subseteq \mathsf{COD(SL}_f)$ ([PRS98, Corollary 4.10]) and $\mathsf{REG} \subseteq \mathsf{COD(SL}_{pf})$.

We illustrate these constructions in an example.

Example 7.4 Consider the finite automaton $\mathcal{A} = (\{s,t,f\}, \{a,b\}, \{(s,b,t), (t,a,t), (t,b,f)\}, s, \{f\})$, for which $L(\mathcal{A}) = ba^*b$. We define the alphabet Σ as

$$\Sigma = \{s,t,f\} \times \{a,b\} \times \{1,2\},$$

the elements of which we write as, e.g., $ta2$ rather than as $(t,a,2)$. The sticker system $\gamma = (\Sigma, id, D_u, D_\ell, A)$ is constructed as follows:

$$\begin{aligned} D_u &= \{ta2\,ta1\,,\ ta2\,tb1\,,\ tb2\} \\ D_\ell &= \{ta1\,ta2\,,\ ta1\,tb2\,,\ tb1\} \\ A &= \{(sb1\,,\ sb1\,ta2)\,,\ (sb1\,,\ sb1\,tb2)\} \end{aligned}$$

and the coding $h : \Sigma \to \{a,b\}$ is defined by $h(qci) = c$ for each $q \in \{s,t,f\}$, $c \in \{a,b\}$ and $i \in \{1,2\}$.

Note that the stickers of γ are either encodings of two consecutive transitions of $\mathcal{A}$, or encodings of transitions of $\mathcal{A}$ that end in a final state (the latter are needed to stop the computation of γ). Furthermore, note that because of the use of symbols 1 and 2 the computations of γ are intrinsically primitive. These two observations together guarantee that only encodings of paths through the automaton are generated.

Four sample computations of γ are

$$\begin{array}{llll} [sb1\rangle\langle tb2\rangle & [sb1\rangle\langle ta2\ tb1\rangle & [sb1\rangle\langle ta2\,ta1\rangle\langle tb2\rangle & [sb1\rangle\langle ta2\,ta1\rangle\langle ta2\ tb1\rangle \\ [sb1\ \ tb2\rangle & [sb1\ ta2\rangle\langle tb1\rangle & [sb1\ ta2\rangle\,\langle ta1\ tb2\rangle & [sb1\ ta2\rangle\,\langle ta1ta2\rangle\langle tb1\rangle \end{array}$$

Clearly $h(L(\gamma)) = h(L_p(\gamma)) = L(\mathcal{A})$.

Note that each computation of γ that uses the lower sticker $tb1$ to finish is fair, and that all other computations (that use the upper sticker $tb2$ to finish) have one more upper sticker than lower stickers. To guarantee that for each computation of $\mathcal{A}$ there is a fair computation in the sticker system, we only have to add the axioms $(sb1\,tb2, sb1\,tb2)$ and $(sb1\,ta2\,ta1, sb1\,ta2)$ to A. For the new sticker system γ' we then have $h(L_f(\gamma')) = h(L_{pf}(\gamma')) = L(\mathcal{A})$. □

The first five observations above and the fact that REG is closed under codings yield the following propositions.

Proposition 7.1 $\mathsf{SL} \subset \mathsf{REG} = \mathsf{COD(SL)}$

Proposition 7.2 $\mathsf{SL}_p \subseteq \mathsf{REG} = \mathsf{COD}(\mathsf{SL}_p)$

Seventh, contrary to primitivity, which is a local property of computations, fairness is more of a global property, that allows one to count. Consequently it is not surprising that $\mathsf{SL}_f - \mathsf{REG} \neq \varnothing$; an example of a non-regular fair sticker language is $L_f(\gamma) = \{x \in \{aa,bb\}^* \mid \#_a(x) = \#_b(x)\}$ from Example 7.2.

Finally, the smallest upper bound for SL_f that has been established until now is MAT^λ, the family of context-free matrix languages with arbitrary rules ([PRS98, Theorem 4.14]), which is a rather large family containing non-semilinear languages, and which is known to be a strict subset of RE ([RS97]).

7.4 Research topics

In [KP+98] it is demonstrated that the family of fair sticker languages contains non-regular languages, while the family of languages generated by context-free matrix grammars with arbitrary rules is given as an upper bound. In connection with this rather large upper bound the following problem is formulated: "is the family [of fair sticker languages] included in the family of context-free languages (or even in the family of linear languages)?".

In Chapter 8 we answer this question by giving a fair sticker language that is non-linear, while demonstrating that the fair sticker languages are strictly included in another subfamily of the context-free languages, the blind one-counter languages (Theorem 8.1 and Lemma 8.2).

The main result of that chapter is that the connection between these two families is quite strong: blind one-counter languages can be characterized as codings of fair sticker languages (Theorem 8.5), giving the 'fair version' of Proposition 7.1.

From Propositions 7.1 and 7.2 we can derive that $\mathsf{SL} \subset \mathsf{REG} \subseteq \mathsf{COD}(\mathsf{SL}_p)$, or in words: each sticker language is a coding of a primitive sticker language. In Section 9.1 we elaborate on this 'primitive normal form' for sticker systems: we give a direct construction (i.e., not via finite automata) and we extend this construction to fair sticker languages (i.e., we give a direct proof that each fair sticker language is a coding of a primitive fair sticker language). In both our constructions, *every* computation in the resulting sticker system is primitive.

Furthermore, research described in the literature is mostly about comparing different kinds of sticker *systems* (e.g., unrestricted, bidirectional, simple, one-sided, regular; they differ in the kind of axioms and stickers that are allowed) to each other and to the language families from the Chomsky hierarchy. Until now, there has been no attempt to relate the different kinds of sticker *languages* (fair, primitive, etcetera; they are the result of restrictions on the computations of the sticker system) that can be generated by one kind of sticker system. In Sections 9.2 through 9.4 we describe the results of our research in that direction for the unrestricted, fair, primitive and primitive fair sticker languages generated by the type of sticker system that we consider: the simple regular sticker systems.

Chapter 8

Fair sticker languages

We prove that each fair sticker language is accepted by a blind one-counter automaton. Moreover, we show that each blind one-counter language is a coding of a fair sticker language.

8.1 Fair sticker languages are BCA-languages

We answer the question left open in [KP$^+$98, p. 419]: is the family of fair sticker languages included in the family of context-free languages (or even in the family of linear languages)? To start, observe that the language $L_f(\gamma) = \{x \in \{aa, bb\}^* \mid \#_a(x) = \#_b(x)\}$ from Example 7.2 is context-free, but not linear. The non-linearity of $L_f(\gamma)$ can be proved using the pumping lemma for linear languages [HU79, Exercise 6.11], which says that if K is linear, then there is a constant n such that every $z \in K$ with $|z| > n$ can be written as $z = uvwxy$ with $|uvxy| \leq n$, $|vx| \geq 1$ and $uv^iwx^iy \in K$ for all $i \geq 0$. In the case of $L_f(\gamma)$, it is clear that there are no such u, v, w, x, y for $z = a^{2n}b^{4n}a^{2n} \in L_f(\gamma)$.

We will now give a first answer to the question posed in [KP$^+$98], by proving that every fair sticker language is a BCA-language (see Section 2.4), hence context-free.

Theorem 8.1 $\mathsf{SL}_f \subseteq \mathsf{1BCA}$

Proof. Let $\gamma = (\Sigma, \rho, D_u, D_\ell, A)$. Because of Theorem 7.3 we may assume that $\rho = id$. For each $(x_0, y_0) \in A$, construct two BCA's: $\mathcal{B}_{x_0}$ and $\mathcal{B}_{y_0}$, as follows. We describe the construction of $\mathcal{B}_{x_0} = (Q, \Sigma, \delta, q_0, \{f\})$ in detail, $\mathcal{B}_{y_0}$ can be made in an analogous way.

If $x_0 = \lambda$, then $q_0 = f$. If $x_0 \neq \lambda$, then $\mathcal{B}_{x_0}$ has a path labelled by x_0 from its initial to its final state. In both cases the counter is not changed, since the axioms do not have to be counted. Moreover, for each $w \in D_u$, let $\mathcal{B}_{x_0}$ have a (new) path labelled with w from its final to its final state and add 1 to the

counter at one moment somewhere along this path. Note that $L(\mathcal{B}_{x_0}) = \{x_0\}$, which does not seem very useful yet.

Now we construct from each pair of BCA's $\mathcal{B}_{x_0}$ and $\mathcal{B}_{y_0}$, for $(x_0, y_0) \in A$, a BCA $\mathcal{B}_{x_0,y_0}$ for which $L(\mathcal{B}_{x_0,y_0}) = L_f(\gamma_{x_0,y_0})$, where γ_{x_0,y_0} is defined as $(\Sigma, id, D_u, D_\ell, \{(x_0, y_0)\})$, using a slightly adapted version of the product construction for the intersection of two regular languages: for each pair of instructions (p, a, ε, q) in $\mathcal{B}_{x_0}$ and (r, a, ε', s) in $\mathcal{B}_{y_0}$, the BCA $\mathcal{B}_{x_0,y_0}$ contains the instruction $(\langle p, r\rangle, a, \varepsilon - \varepsilon', \langle q, s\rangle)$.

Finally, it is clear that $L_f(\gamma) = \bigcup_{(x_0,y_0)\in A} L(\mathcal{B}_{x_0,y_0})$ is in 1BCA, since 1BCA is closed under union and A is finite. □

Omitting the counter from the previous proof, one constructs a finite state automaton for $L(\gamma) = \bigcup_{(x_0,y_0)\in A}(x_0 \cdot D_u^* \cap y_0 \cdot D_\ell^*)$. This elementary observation shows that $\mathsf{SL} \subseteq \mathsf{REG}$.

The inclusion $\mathsf{SL}_f \subseteq \mathsf{1BCA}$ is strict because ba^*b is not a fair sticker language, whereas $ba^*b \in \mathsf{REG} \subseteq \mathsf{1BCA}$.

Lemma 8.2 $ba^*b \notin \mathsf{SL}_f$

Proof. We reconsider the proof of $ba^+b \notin \mathsf{SL}$, cf. [PR98, Theorem 10] and page 82 of this thesis. Assume that ba^*b is the fair language of a sticker system $\gamma = (\{a, b\}, \rho, D_u, D_\ell, A)$. According to Theorem 7.3 we may assume that $\rho = id$. Let $D_u \cap a^+ = \{x_1, \ldots, x_m\}$ and $D_\ell \cap a^+ = \{y_1, \ldots, y_n\}$ be the sets of stickers consisting of a's only. Every string ba^ib that is longer than the axioms can be decomposed as $\alpha_u x_1^{j_1} \ldots x_m^{j_m} \beta_u = \alpha_\ell y_1^{k_1} \ldots y_n^{k_n} \beta_\ell$, with α_u the upper part of an axiom (or a string from D_u starting with b), and $\beta_u \in D_u$ ending in b, and similarly for α_ℓ, β_ℓ. The vector $\nu_i = (j_1, \ldots, j_m, k_1, \ldots, k_n)$ assigns to ba^ib the number of each sticker containing only a's occurring in a possible decomposition of the upper and the lower strand.

Because we have only a finite number of choices, an infinite number of ba^ib have the same strings $\alpha_u, \alpha_\ell, \beta_u, \beta_\ell$ in their decompositions. According to Dickson's lemma [Dic13, Lemma B] we can find ba^ib and $ba^{i'}b$ $(i' > i)$ in this infinite sequence such that $\nu_{i'} \geq \nu_i$ (componentwise). Now the vector $\nu_{i'} - \nu_i$ defines a 'fair decomposition' of $a^{i'-i}$, which shows that $ba^iba^{i'-i} \in L_f(\gamma)$, contradicting $L_f(\gamma) = ba^*b$. □

In the next section we make our answer more precise, in the sense that we show that 1BCA is a rather close upper bound for SL_f: every BCA-language is a coding of a fair sticker language.

8.2 BCA-languages are codings of fair sticker languages

In the case of arbitrary, i.e., not necessarily fair, sticker languages the simulation of sticker systems by finite automata can be reversed provided that one can use

a coding (Proposition 7.1). In this section we demonstrate that Proposition 7.1 can be extended to fair sticker languages and BCA-languages: every language in 1BCA is the coding of a fair sticker language (Theorem 8.5). First we illustrate this in Example 8.1. Then we show that, for a particular kind of BCA called *sticky*, this example can be generalised (Lemma 8.3). Finally, we explain how these sticky BCA's can be used to construct a coding of a fair sticker language for every BCA-language.

Example 8.1 Consider the BCA $\mathcal{A}$ given below.

The automaton accepts the language $\{\ w \in \{ab, ac\}^* \mid \#_b(w) = \#_c(w)\ \}$. This can be verified by considering the four-letter segments *abab*, *abac*, *acab*, and *acac*. While the automaton makes a cycle on these segments (starting and ending in $\{b_0, c_0\}$), it changes its counter by -1, 0, 0, and $+1$, respectively. Note that there is a much easier BCA that accepts the same language; however, the special structure of the automaton above is essential for this example.

First, we forget about the counter, and we have a look at the finite-state behaviour of $\mathcal{A}$, $\{abab, abac, acab, acac\}^*$. A computation of $\mathcal{A}$ can be simulated by a (fair) computation of a sticker system with overlapping stickers, cf. [KP$^+$98, Lemma 5], illustrated as follows, with brackets to delimit the stickers and the axiom:

$$\begin{array}{l} [\rangle\langle a_1\ b_2\ \ a_3\ b_0\rangle\langle a_1\ b_2\ \ a_3\ c_0\rangle\langle a_1\ c_2\ \ a_3\ b_0\rangle\langle a_1\ c_2\ \ a_3\ c_0\rangle \\ [\ \ a_1\ b_2\rangle\langle a_3\ b_0\ \ a_1\ b_2\rangle\langle a_3\ c_0\ \ a_1\ c_2\rangle\langle a_3\ b_0\ \ a_1\ c_2\rangle\langle a_3 c_0\rangle \end{array}$$

Second, we can include the contents of the counter by representing it as the difference between the number of upper and lower stickers in the computation of the sticker system. For each increment instruction we detach the last component of an upper sticker, and similarly for decrement instructions and lower stickers.

$$\begin{array}{l} \qquad\qquad\qquad\qquad\qquad\quad +1 \qquad\qquad\qquad\qquad\qquad\quad +1 \\ [\rangle\langle a_1\ \ b_2\ \ a_3\ b_0\rangle\langle a_1\ \ b_2\ \ a_3\rangle\langle c_0\rangle\langle a_1\ c_2\ \ a_3\ b_0\rangle\langle a_1\ c_2\ a_3\rangle\langle c_0\rangle \\ [\ \ a_1\rangle\langle b_2\rangle\langle a_3\ b_0\ a_1\rangle\langle b_2\rangle\langle a_3\ \ c_0\ \ a_1\ c_2\rangle\langle a_3\ b_0\ \ a_1\ c_2\rangle\langle a_3\ c_0\rangle \\ \qquad\ \text{-1} \qquad\qquad\ \ \text{-1} \end{array}$$

Let $\gamma = (Q, id, D_u, D_\ell, A)$ be the sticker system specified by

$$\begin{array}{rcl} A & = & \{\ (\lambda,\lambda)\ ,\ (\lambda,a_1)\ ,\ (\lambda,a_1c_2)\ \} \\ D_u & = & \{a_1b_2a_3\ ,\ a_1b_2a_3b_0\ ,\ a_1c_2a_3\ ,\ a_1c_2a_3b_0\ ,\ c_0\} \\ D_\ell & = & \{a_3b_0a_1\ ,\ a_3b_0a_1c_2\ ,\ a_3c_0a_1\ ,\ a_3c_0a_1c_2\ ,\ a_3b_0\ ,\ a_3c_0\ ,\ b_2\}. \end{array}$$

Then $L(\mathcal{A})$ is obtained by applying to $L_f(\gamma)$ the coding $h : Q \to \{a,b,c\}$ that maps a_1, a_3 to a, b_0, b_2 to b, and c_0, c_2 to c. □

Note that, in our construction, paths through the automaton are simulated in the sticker system by building them from segments of four consecutive states, instead of two consecutive states, as is the case in [KP+98, Lemma 5]. The reason for this is that we sometimes want to disconnect the last state from such a segment. If we use segments of length shorter than four, then it becomes possible to generate sequences of states that do no form a path in the automaton (see also the proof of Lemma 8.3).

A crucial property of the BCA from the above example is formalized in the following notion.

Definition 8.1 Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a BCA. It is *sticky* if there is a partition of its state set $Q = \bigcup_{i=0}^{3} Q_i$ such that δ is a subset of

$$\begin{array}{lcl} (Q_0 \times \Sigma \times \{0\} \times Q_1) & \cup & (Q_1 \times \Sigma \times \{-1,0\} \times Q_2) \quad \cup \\ (Q_2 \times \Sigma \times \{0\} \times Q_3) & \cup & (Q_3 \times \Sigma \times \{0,+1\} \times Q_0) \end{array}$$

and such that $q_0 \in Q_0$ and $F \subseteq Q_0$. □

The BCA $\mathcal{A}$ from Example 8.1 is sticky, since obviously the partition $Q_0 = \{b_0, c_0\}$, $Q_1 = \{a_1\}$, $Q_2 = \{b_2, c_2\}$ and $Q_3 = \{a_3\}$ satisfies the requirements.

A sticky BCA changes its counter in a very restrictive way: in each segment of four instructions the automaton may increment and decrement its counter only once, and only at specific positions. Note that the language accepted by a sticky BCA always consists of strings with lengths that are multiples of four.

We generalise the construction from Example 8.1 to sticky BCA's.

Lemma 8.3 *Let $\mathcal{A}$ be a sticky* BCA. *Then there exist a sticker system γ and a coding h such that $L(\mathcal{A}) = h(L_f(\gamma))$.*

Proof. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a sticky BCA. We write the state set as a disjoint union $Q = \bigcup_{i=0}^{3} Q_i$ as in Definition 8.1.

Let $h : Q \to \Sigma$ be a coding such that each instruction is of the form $(p, h(q), \varepsilon, q)$, i.e., all instructions ending in a given state read the same letter. This can easily be achieved by splitting states into several copies – one for each

letter from the alphabet, each of which has the same outgoing instructions – and re-routing the instructions into the appropriate copy. In the same vein we assume that there exists a partition $Q_2 = Q_2^0 \cup Q_2^-$, such that each instruction (p, a, ε, q) entering Q_2^0 (Q_2^-) has $\varepsilon = 0$ ($\varepsilon = -1$, respectively). Similarly we assume $Q_0 = Q_0^0 \cup Q_0^+$.

Construction. A sticker system $\gamma = (Q, id, D_u, D_\ell, A)$ with $h(L_f(\gamma)) = L(\mathcal{A})$ is constructed as follows. To avoid confusion between states and stickers, we keep the intuitive bracket notation for the stickers.

upper stickers. For every pair of consecutive instructions $(p_1, a_2, \varepsilon_2, p_2)$ and $(p_2, a_3, 0, p_3)$ with $p_1 \in Q_1$, D_u contains the stickers $\langle p_1p_2p_3\rangle$ and, for every $p_0 \in Q_0^0$, $\langle p_1p_2p_3p_0\rangle$. For each $p_+ \in Q_0^+$, D_u contains the sticker $\langle p_+\rangle$.

lower stickers. For every pair of consecutive instructions $(p_3, a_0, \varepsilon_0, p_0)$ and $(p_0, a_1, 0, p_1)$ with $p_3 \in Q_3$, D_ℓ contains the stickers $\langle p_3p_0p_1\rangle$ and, for every $p_2 \in Q_2^0$, $\langle p_3p_0p_1p_2\rangle$. For each $p_- \in Q_2^-$, D_ℓ contains the sticker $\langle p_-\rangle$. For every instruction $(p_3, a_0, \varepsilon_0, p_0)$ with $p_3 \in Q_3$, $p_0 \in F$, D_ℓ contains the sticker $\langle p_3p_0\rangle$.

axioms. For every instruction $(q_0, a_1, 0, p_1)$ with $p_1 \in Q_1$, A contains the pairs (λ, p_1) and, for every $p_2 \in Q_2^0$, (λ, p_1p_2). If $q_0 \in F$, i.e., $\lambda \in L(\mathcal{A})$, then (λ, λ) is added to A.

Correctness. Observe that $\lambda \in L(\mathcal{A})$ if and only if $\lambda \in L_f(\gamma)$ if and only if $\lambda \in h(L_f(\gamma))$.

Now, let $\pi = p_1p_2p_3 \dots p_n \in Q^+$ be an element of $L_f(\gamma)$, for some $n \geq 1$.

First, we reconstruct a computation of $\mathcal{A}$ by following the computation of π in γ. Since there is no computation longer than (λ, λ) starting with $(\lambda, \lambda) \in A$ – all stickers in D_u start with symbols from $Q_1 \cup Q_0^+$, whereas all stickers from D_ℓ start with symbols from $Q_3 \cup Q_2^-$ – we know that the computation of π in γ started either with $(\lambda, p_1) \in A$ or with $(\lambda, p_1p_2) \in A$, where $p_1 \in Q_1$. According to the construction of A, δ contains an instruction $(q_0, a_1, 0, p_1)$.

We continue by observing that each upper sticker of length 3 or 4 starts at position $4i+1$, and that each lower sticker of length 2, 3 or 4 starts at position $4i+3$, for some $i \geq 0$. It is easy to see that this follows from the only possible computation of π, here illustrated for $n = 8$:

$$\begin{array}{l} [\rangle\langle\ p_1 \to p_2 \to p_3 \cdots p_4\ \rangle\langle\ p_5 \to p_6 \to p_7 \cdots p_8\rangle \\ [\quad p_1 \cdots p_2\ \rangle\langle\ p_3 \to p_4 \to p_5 \cdots p_6\ \rangle\langle\ p_7 \to p_8\rangle \end{array}$$

Here the arrows indicate parts of a sticker that represent instructions from δ, while the dotted lines do not necessarily correspond to an instruction from δ and, at the same time, indicate that the next symbol may be detached to form a sticker of length 1. Moreover, observing D_u we find instructions

$(p_{4i+1}, a_{4i+2}, \varepsilon_{4i+2}, p_{4i+2})$ and $(p_{4i+2}, a_{4i+3}, 0, p_{4i+3})$, while D_ℓ gives rise to instructions $(p_{4i+3}, a_{4i+4}, \varepsilon_{4i+4}, p_{4i+4})$ and $(p_{4i+4}, a_{4i+5}, 0, p_{4i+5})$.

Since p_n is the last symbol of stickers from both D_u and D_ℓ, we know that $p_n \in F \subseteq Q_0$, and there exists an instruction $(p_{n-1}, a_n, \varepsilon_n, p_n)$ in δ. Note that n is a multiple of four, and we write $n = 4k$.

Second, we address the matter of fairness. To compute the contents of the counter we study the even positions of π. Observe that $\varepsilon_{4i+4} = +1$ if and only if $p_{4i+4} \in Q_0^+$, which implies that the sticker $\langle p_{4i+4} \rangle$ is used in the upper part of the solution. Otherwise, if $\varepsilon_{4i+4} = 0$, then p_{4i+4} is the fourth element of the sticker $\langle p_{4i+1} p_{4i+2} p_{4i+3} p_{4i+4} \rangle$. Thus, the number of upper stickers equals $k + \sum_{i=0}^{k-1} \varepsilon_{4i+4}$. Similarly, the number of lower stickers equals $k - \sum_{i=0}^{k-1} \varepsilon_{4i+2}$. Consequently, fairness of the sticker solution is equivalent to counter value zero and acceptance by the BCA.

The above shows that $h(L_f(\gamma)) \subseteq L(\mathcal{A})$. For the converse inclusion $L(\mathcal{A}) \subseteq h(L_f(\gamma))$ a similar reasoning can be given. □

Sticky BCA's form a normal form for BCA's accepting languages consisting of strings with lengths that are multiples of four. The idea behind this is the following.

Let $\mathcal{A}$ be a BCA, and suppose that we want to construct a sticky BCA $\mathcal{B}$ such that $L(\mathcal{A}) = L(\mathcal{B})$. In every four steps, $\mathcal{A}$ changes the contents of its counter by at most ± 4. The new BCA $\mathcal{B}$ however, may change its counter by at most ± 1 in the corresponding four steps. To make up for this, we change the interpretation of the counter value of $\mathcal{B}$: each unit on the counter of $\mathcal{B}$ represents 4 units on the counter of $\mathcal{A}$, an idea known at least since [FMR68]. Now, $\mathcal{B}$ simulates the computation of $\mathcal{A}$. Each change made to the counter of $\mathcal{A}$ is recorded in the finite-state memory of $\mathcal{B}$. Only when allowed (at the specific points in the four step cycle), $\mathcal{B}$ moves any excess of ± 4 units of $\mathcal{A}$'s counter as one unit to (or from) its own counter.

Lemma 8.4 *For each* BCA *that accepts only strings of lengths that are multiples of four, there exists an equivalent sticky* BCA*.*

Proof. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a BCA as mentioned in the lemma. We construct a sticky BCA $\mathcal{B}$ such that $L(\mathcal{A}) = L(\mathcal{B})$.

Let $I = \{0, 1, 2, 3\}$. The state set Q' of $\mathcal{B}$ equals

$$Q \times I \times \{-4, -3, -2, -1, 0, 1, 2, 3\},$$

the elements of which we denote as $p.i.m$, rather than as (p, i, m). Here $p \in Q$ represents the state of $\mathcal{A}$, $i \in I$ keeps track of the four step cycle, and $-4 \leq m \leq 3$ is the remainder value of $\mathcal{A}$'s counter not yet stored in the counter of $\mathcal{B}$. Hence, if c is the value of $\mathcal{A}$'s counter and c' the value of $\mathcal{B}$'s counter, then the equality $c = 4c' + m$ should hold for each pair of corresponding instantaneous

descriptions of $\mathcal{A}$ and $\mathcal{B}$. The initial state of $\mathcal{B}$ equals $q_0.0.0$, its final state set equals $F \times \{0\} \times \{0\}$.

Let (p, a, ε, q) be an instruction of $\mathcal{A}$. Then $\mathcal{B}$ has the instructions

$$\begin{array}{ll}
(\ p.0.m,\ a,\ 0,\ q.1.m{+}\varepsilon\ ) & \\
(\ p.1.m,\ a,\ -1, q.2.m{+}\varepsilon{+}4\ ) & \text{if} \quad m+\varepsilon < -1 \\
(\ p.1.m,\ a,\ 0,\ q.2.m{+}\varepsilon\ ) & \text{if} \quad m+\varepsilon \geq -1 \\
(\ p.2.m,\ a,\ 0,\ q.3.m{+}\varepsilon\ ) & \\
(\ p.3.m,\ a,\ +1, q.0.m{+}\varepsilon{-}4\ ) & \text{if} \quad m+\varepsilon \geq 1 \\
(\ p.3.m,\ a,\ 0,\ q.0.m{+}\varepsilon\ ) & \text{if} \quad m+\varepsilon < 1
\end{array}$$

We chose to check the relation between $m{+}\varepsilon$ and ± 1 rather than between $m{+}\varepsilon$ and ± 4, although the latter seems more logical. The reason for this is that we need to prevent the occurrence of the situation where $i = 0$ and $c = 4c' + m = 0$ while $c' \neq 0$ and $m \neq 0$ (which can occur only when m is a multiple of 4), i.e., $\mathcal{B}$ does not accept while it should. Because of this choice, indeed the reachable configurations of $\mathcal{B}$ satisfy the following restrictions, for $p.i.m \in Q'$:

$$\begin{array}{lll}
\text{if} \quad i = 0 & \text{then}\ m \in & \{-3,-2,-1,0\} \\
\phantom{\text{if} \quad i = }1 & & \{-4,-3,-2,-1,0,1\} \\
\phantom{\text{if} \quad i = }2 & & \{-1,0,1,2\} \\
\phantom{\text{if} \quad i = }3 & & \{-2,-1,0,1,2,3\}
\end{array}$$

It is easy to see that $\mathcal{B}$ is sticky, as it adheres to the four step cycle from Definition 8.1.

Moreover, our construction introduces for each instruction (p, a, ε, q) of $\mathcal{A}$ exactly one instruction $(p.i.m, a, \varepsilon', q.i'.m')$ for each pair i, m. This makes it straightforward to show that a computation $(q_0, xy, 0) \vdash^j (q, y, c)$ of $\mathcal{A}$ corresponds with a computation $(q_0.0.0, xy, 0) \vdash^j (q.i.m, y, c')$ of $\mathcal{B}$ satisfying $c = 4c' + m$, and $i = j \bmod 4$.

To show that $L(\mathcal{B}) \subseteq L(\mathcal{A})$, observe that if $\mathcal{B}$ reaches a final state $q.0.0$ with counter value zero, then $\mathcal{A}$ (using the corresponding computation) reaches final state q, also with counter value zero.

Conversely, assume that $\mathcal{A}$ reaches a final state $q \in F$ with counter value zero. Now the corresponding computation of $\mathcal{B}$ reaches some state $q.i.m$ and counter value c' satisfying the invariant $m + 4c' = 0$. By the length restriction of strings accepted by $\mathcal{A}$ we know that $i = 0$. Hence, taking into account the reachable states of $\mathcal{B}$, we have $m \in \{-3, \ldots, 0\}$. Thus $m + 4c' = 0$ implies that $m = 0$ and thus $c' = 0$, corresponding to acceptance with counter value zero in final state $q.0.0$.

A more formal inductive proof that $L(\mathcal{A}) = L(\mathcal{B})$ is left to the reader. □

Finally we arrive at the main result of this chapter, the equivalence of blind one-counter languages and codings of fair sticker languages. Note the similarity with the situation for (arbitrary) sticker languages (Proposition 7.1).

Theorem 8.5 $\mathsf{SL}_f \subset \mathsf{1BCA} = \mathsf{COD}(\mathsf{SL}_f)$.

Proof. By Theorem 8.1, $\mathsf{SL}_f \subseteq \mathsf{1BCA}$. The inclusion is strict by Lemma 8.2. As $\mathsf{1BCA}$ is closed under codings, the inclusion $\mathsf{COD}(\mathsf{SL}_f) \subseteq \mathsf{1BCA}$ follows. We proceed by proving the converse inclusion.

Let $L \in \mathsf{1BCA}$. For every string w, we define $L_w = \{ x \mid wx \in L, |x| = 0 \bmod 4 \}$. By the closure properties we have established for $\mathsf{1BCA}$ in Section 2.4, L_w is also in $\mathsf{1BCA}$, and, by Lemma 8.4, it is accepted by a sticky BCA. Consequently, it is the coding of a fair sticker language (Lemma 8.3).

Note that $L = \bigcup_{|w| \leq 3} w \cdot L_w$. A sticker system for the language $w \cdot L_w$ is obtained from the one for L_w by replacing each axiom (x, y) by (wx, wy) and extending the used coding with the identity on the alphabet of L. Assuming the sticker systems representing the $w \cdot L_w$ have disjoint alphabets (by renaming), we build a sticker system for L by taking their (finite) union. □

Our characterization shows that $\mathsf{COD}(\mathsf{SL}_f)$ is a more 'robust' family than SL_f itself, comparable to the situation for $\mathsf{COD}(\mathsf{SL})$ and SL. In particular, we can conclude that $\mathsf{COD}(\mathsf{SL}_f)$ enjoys the many closure properties of a principal rational cone (arbitrary morphisms, inverse morphisms, intersection with regular languages, and union). Some of these properties seem to require rather involved proofs, should we want to show them by direct construction.

8.3 Summary

We have answered a question concerning the position of SL_f in the Chomsky hierarchy: is $\mathsf{SL}_f \subseteq \mathsf{CF}$ or even $\mathsf{SL}_f \subseteq \mathsf{LIN}$? We showed that $\mathsf{SL}_f \not\subseteq \mathsf{LIN}$, and we proved that $\mathsf{SL}_f \subset \mathsf{1BCA}$, which is a subfamily of CF. Moreover, we showed that $\mathsf{1BCA}$ is a rather close upper bound for SL_f by proving that $\mathsf{1BCA} = \mathsf{COD}(\mathsf{SL}_f)$.

Chapter 9

A hierarchy of sticker families

We give direct constructions to create for each (fair) sticker system an equivalent – modulo a coding – (fair) sticker system that can do only primitive computations. We also investigate the relations between the families of unrestricted, primitive, fair and primitive fair sticker languages.

9.1 A primitive 'normal form'

In [PRS98, Chapter 4] the following proposition can be derived from Theorem 4.1, Lemma 4.1 and Corollary 4.7 to Theorem 4.8 (see also Section 7.3 of this thesis):

Proposition 9.1 $\mathsf{SL} \subseteq \mathsf{REG} = \mathsf{COD}(\mathsf{SL}_p)$

This implies that for every sticker system γ a sticker system γ' and a coding h can be constructed such that $L(\gamma) = h(L_p(\gamma'))$, or in words: every language that is generated by a sticker system can also be generated using only primitive computations, provided that one is allowed to use a coding. A closer look at the proof of [PRS98, Theorem 4.8] reveals that γ' can be constructed such that *each* of its computations is primitive. However, this construction is not direct: first an equivalent regular grammar is created for the original sticker system (using the construction given in the proof of [PRS98, Theorem 4.1]), and then, using a coding, this regular grammar is translated into a sticker system that allows only primitive computations (see the proof of [PRS98, Theorem 4.8]).

We show that there is also an elegant direct construction (Lemma 9.1), based on the observation that if there is no pair consisting of an upper and a lower sticker such that their first letters are complementary, then every computation is intrinsically primitive, since it cannot continue after a blunt end. (The same observation is used in the proof of Lemma 8.3 and in Example 7.4, but for different reasons.) To cover also the non-primitive computations of the

original system, we add new stickers that are concatenations of two old stickers. These new stickers can now be used at the positions where in the original computation an intermediate blunt end occurred.

As an aside, note that if all computations of a sticker system γ are primitive, then $L_p(\gamma) = L(\gamma)$, but not the other way around: some words in $L(\gamma)$ may have both a primitive and a non-primitive computation.

Lemma 9.1 *For every sticker system γ a sticker system γ' and a coding h can be constructed such that all computations of γ' are primitive and $h(L(\gamma')) = L(\gamma)$.*

Proof. Let $\gamma = (\Sigma, \rho, D_u, D_\ell, A)$ be a sticker system. We may assume that ρ equals the identity on Σ.

Construction. Define $\gamma' = (\Sigma', \rho', D'_u, D'_\ell, A')$ as follows:

$$\begin{aligned}
\Sigma' &= \Sigma \cup \{\overline{a} \mid a \in \Sigma\} \\
\rho' &= \rho \cup \{(a, \overline{a}), (\overline{a}, a) \mid a \in \Sigma\} \\
D'_u &= \{\overline{a_1}\, a_2 \ldots a_k \mid a_1 a_2 \ldots a_k \in D_u \text{ for } a_1, \ldots, a_k \in \Sigma\} \cup \\
&\quad\; \{\overline{a_1}\, a_2 \ldots a_k x \mid a_1 a_2 \ldots a_k\,,\ x \in D_u \text{ for } a_1, \ldots, a_k \in \Sigma\} \\
D'_\ell &= \{\overline{a_1}\, a_2 \ldots a_k \mid a_1 a_2 \ldots a_k \in D_\ell \text{ for } a_1, \ldots, a_k \in \Sigma\} \cup \\
&\quad\; \{\overline{a_1}\, a_2 \ldots a_k y \mid a_1 a_2 \ldots a_k\,,\ y \in D_\ell \text{ for } a_1, \ldots, a_k \in \Sigma\} \\
A' &= A \cup \{(x_0 x, y_0) \mid (x_0, y_0) \in A \text{ and } x \in D_u\}
\end{aligned}$$

The coding h is defined by $h(a) = h(\overline{a}) = a$ for all $a \in \Sigma$.

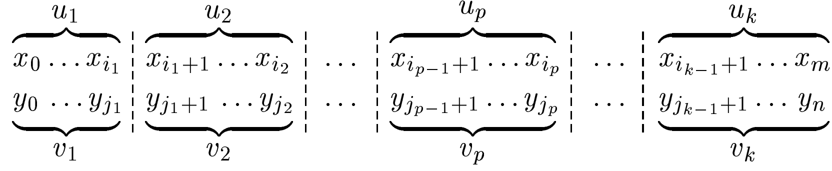
We call each element of D'_u (D'_ℓ) that is a concatenation – modulo the coding h – of two elements from D_u (D_ℓ) an upper (lower) *double-sticker.*

Correctness. First, it is easy to see that all computations in γ' are primitive: since we have marked the first letter of each sticker and have defined ρ' in such a way that two marked letters are never complementary, it is clear that no complete computation can be prolonged.

Second, we prove that $L(\gamma) \subseteq h(L(\gamma'))$. Let $x \in L(\gamma)$, i.e., $x = x_0 x_1 \ldots x_m = y_0 y_1 \ldots y_n$, with $(x_0, y_0) \in A$, $x_1, \ldots, x_m \in D_u$ for some $m \geq 0$, and $y_1, \ldots, y_n \in D_\ell$ for some $n \geq 0$.

We start by observing that each element of D_u, D_ℓ or A is – modulo the coding h – also in D'_u, D'_ℓ or A', respectively. Hence if the computation $(x_0 x_1 \ldots x_m, y_0 y_1 \ldots y_n)$ of γ is primitive, then there is a primitive computation $(x_0 x'_1 \ldots x'_m, y_0 y'_1 \ldots y'_n)$ of γ', with $h(x'_i) = x_i$ for $1 \leq i \leq m$ and $h(y'_j) = y_j$ for $1 \leq j \leq n$.

Now assume that $(x_0 x_1 \ldots x_m, y_0 y_1 \ldots y_n)$ is not primitive. Let $i_1, \ldots, i_k$ and $j_1, \ldots, j_k$, with $k \geq 2$, $0 \leq i_1 < \ldots < i_{k-1} < i_k = m$ and $0 \leq j_1 < \ldots < j_{k-1} < j_k = n$, be all indices such that $(x_0 \ldots x_{i_p}, y_0 \ldots y_{j_p})$ is a complete computation of γ, for p such that $1 \leq p \leq k$. Define $u_1 = x_0 \ldots x_{i_1}$, $v_1 = y_0 \ldots y_{j_1}$, $u_p = x_{i_{p-1}+1} \ldots x_{i_p}$ and $v_p = y_{j_{p-1}+1} \ldots y_{j_p}$ for each $2 \leq p \leq k$.



Hence there are exactly $k-1$ intermediate blunt ends in the computation $(x_0x_1\dots x_m, y_0y_1\dots y_n)$, and they are located on the borders between $(u_1\dots u_p, v_1\dots v_p)$ and $(u_{p+1}\dots u_k, v_{p+1}\dots v_k)$, for each $1\leq p\leq k-1$.

For every odd p, with $1\leq p\leq k-1$, u_p ends with x_{i_p} and u_{p+1} starts with x_{i_p+1}. According to the definition of D'_u, there is a double-sticker $x'_{i_p}\cdot x_{i_p+1}$ in D'_u with $h(x'_{i_p})=x_{i_p}$. (If $p=1$ and $i_1=0$, then there is an axiom (x_0x_1, y_0) in A'.)

Analogously, for every even p, with $2\leq p\leq k-1$, v_p ends with y_{j_p}, v_{p+1} starts with y_{j_p+1} and D'_ℓ contains the double-sticker $y'_{j_p}\cdot y_{j_p+1}$ with $h(y'_{j_p})=y_{j_p}$.

Note that in this way at every 'odd' intermediate blunt end an upper double-sticker is used, while at every 'even' intermediate blunt end a lower double-sticker is used. This alternation of upper and lower double-stickers solves any problems that may arise when three consecutive intermediate blunt ends are separated only by one upper and lower sticker between the first and second blunt end, and one upper and lower sticker between the second and third. In other words, this alternation guarantees that we do not need concatenations of more than two 'old' stickers (modulo the coding).

Consequently $x\in h(L(\gamma'))$ and $L(\gamma)\subseteq h(L(\gamma'))$.

Finally, it is obvious that every upper sticker of γ' has a one-to-one correspondence (via the coding h) with either an upper sticker of γ or the concatenation of two upper stickers of γ, and similarly for the lower stickers. Furthermore, every axiom of γ' is either an axiom from γ or an axiom from γ in which the first component has an upper sticker from γ concatenated to it. Therefore it is straightforward that $h(L(\gamma'))\subseteq L(\gamma)$. □

In the sequel of this section we demonstrate that Lemma 9.1 can be extended to *fair* sticker languages. As before, the result is already known, but only through an indirect construction that can be derived from results in the previous chapter: combining the constructions in the proofs of Theorem 8.5 and Lemma 8.3 we get the following result.

Lemma 9.2 $\mathsf{SL}_f\subset \mathsf{1BCA}\subseteq \mathsf{COD}(\mathsf{SL}_{pf})$

We give here a direct construction, in which we use essentially the same technique as in the proof of Lemma 9.1 above, i.e., we mark the beginning of each sticker and we use double-stickers to 'bridge' (or 'ligate') intermediate blunt ends.

Note that every time an upper (lower) double-sticker is used instead of two normal upper (lower) stickers, the total number of upper (lower) stickers used in the computation decreases by 1. Since we now want (un)fair computations to remain (un)fair, we have to guarantee that exactly as many double-stickers are used in the upper strand as in the lower strand. This can be done by ensuring that double-stickers are applied alternately in the upper strand and in the lower strand at the positions where in the original system an intermediate blunt end occurred. In the previous proof we already used the fact that the application of double-stickers *can* happen alternately, and it is easy to extend the method used there to guarantee this alternating application: add a symbol to each letter in the alphabet, indicating in which strand the next double-sticker should occur (say that '$\uparrow$' means upper strand and '$\downarrow$' means lower strand) and use these extended letters in the stickers and axioms in both strands, requiring that at each position either both strands have $\uparrow$ or both strands have $\downarrow$. Change from $\uparrow$ to $\downarrow$ if an upper double-sticker is used, and from $\downarrow$ to $\uparrow$ if a lower double-sticker is used. Start, e.g., with $\uparrow$.

If the original computation had an even number of intermediate blunt ends, then the above approach gives a new computation in which the difference between the number of upper and the number of lower stickers is the same as in the original computation. However, if the original computation had an odd number of intermediate blunt ends, then the lower strand of the new computation will have one extra sticker less than the other strand. To prevent this from happening, we require that for such an 'odd' original computation the first intermediate blunt end should be bridged in both the upper and the lower strand. We indicate this with the symbol '$\updownarrow$' in both strands.

We use the extra symbol '$*$' to ensure that the translation that is meant for 'even' original computations is not used for 'odd' original computations and vice versa (actually, $*$ ensures that computations do not end with $\downarrow$ or $\updownarrow$), and '#' to guarantee that the $\updownarrow$ symbols have the desired effect (for details see the proof below).

Lemma 9.3 *For every sticker system γ a sticker system γ' and a coding h can be constructed such that all computations of γ' are primitive and $h(L_f(\gamma')) = L_f(\gamma)$.*

Proof. Let $\gamma = (\Sigma, \rho, D_u, D_\ell, A)$. We may assume that ρ is the identity on Σ.

Construction. Define $\gamma' = (\Sigma', \rho', D'_u, D'_\ell, A')$ as follows, where the a_i, b_i, c_i and d_i are symbols in Σ:

$$\begin{aligned}
\Sigma' &= \{a\uparrow,\ a\downarrow,\ \overline{a}\uparrow,\ \overline{a}\downarrow,\ a\downarrow*,\ \overline{a}\downarrow*,\ a\updownarrow,\ \overline{a}\updownarrow,\ a\updownarrow\#,\ \overline{a}\updownarrow\# \ \mid\ a \in \Sigma\,\} \\
\rho' &= \{(a\uparrow, a\uparrow)\,,\ (\overline{a}\uparrow, a\uparrow)\,,\ (a\uparrow, \overline{a}\uparrow)\,,\ (a\downarrow, a\downarrow)\,,\ (\overline{a}\downarrow, a\downarrow)\,,\ (a\downarrow, \overline{a}\downarrow), \\
&\qquad (a\downarrow, a\downarrow*)\,,\ (a\downarrow*, a\downarrow)\,,\ (a\downarrow, \overline{a}\downarrow*)\,,\ (\overline{a}\downarrow*, a\downarrow)\,,\ (\overline{a}\downarrow, a\downarrow*)\,,\ (a\downarrow*, \overline{a}\downarrow), \\
&\qquad (a\updownarrow, a\updownarrow)\,,\ (a\updownarrow\#, a\updownarrow\#) \mid a \in \Sigma\,\}
\end{aligned}$$

$$
\begin{array}{rcl}
D'_u & = & \{\ \overline{a_1}\!\uparrow a_2\!\uparrow \dots a_k\!\uparrow,\\
& & \overline{a_1}\!\downarrow a_2\!\downarrow \dots a_k\!\downarrow\!*,\\
& & \overline{a_1}\!\updownarrow a_2\!\updownarrow \dots a_k\!\updownarrow\!*,\\
& & \overline{a_1}\!\uparrow a_2\!\uparrow \dots a_k\!\uparrow b_1\!\downarrow \dots b_n\!\downarrow\!*,\\
& & \overline{a_1}\!\updownarrow a_2\!\updownarrow \dots a_k\!\updownarrow\!\#b_1\!\uparrow \dots b_n\!\uparrow,\\
& & \overline{a_1}\!\updownarrow a_2\!\updownarrow \dots a_k\!\updownarrow\!\#b_1\!\uparrow \dots b_n\!\uparrow c_1\!\downarrow \dots c_m\!\downarrow\!* \quad |\\
& & \qquad\qquad\qquad a_1a_2\dots a_k\,,\ b_1\dots b_n\,,\ c_1\dots c_m \in D_u\ \}\\
D'_\ell & = & \{\ \overline{a_1}\!\uparrow a_2\!\uparrow \dots a_k\!\uparrow,\\
& & \overline{a_1}\!\downarrow a_2\!\downarrow \dots a_k\!\downarrow\!*,\\
& & \overline{a_1}\!\updownarrow a_2\!\updownarrow \dots a_k\!\updownarrow\!*,\\
& & \overline{a_1}\!\downarrow a_2\!\downarrow \dots a_k\!\downarrow b_1\!\uparrow \dots b_n\!\uparrow,\\
& & \overline{a_1}\!\updownarrow a_2\!\updownarrow \dots a_k\!\updownarrow\!\#b_1\!\uparrow \dots b_n\!\uparrow \quad | \quad a_1a_2\dots a_k\,,\ b_1\dots b_n \in D_\ell\ \}\\
A' & = & \{\ (a_1\!\uparrow \dots a_k\!\uparrow,\ b_1\!\uparrow \dots b_n\!\uparrow) \quad | \quad (a_1\dots a_k\,,\ b_1\dots b_n) \in A\ \} \ \cup\\
& & \{\ (a_1\!\uparrow \dots a_k\!\uparrow c_1\!\downarrow \dots c_m\!\downarrow\!*,\ b_1\!\uparrow \dots b_n\!\uparrow) \ \ |\ \ k \ge n,\\
& & \qquad\qquad (a_1\dots a_k\,,\ b_1\dots b_n) \in A \text{ and } c_1\dots c_m \in D_u\} \ \cup\\
& & \{\ (a_1\!\updownarrow \dots a_k\!\updownarrow,\ b_1\!\updownarrow \dots b_n\!\updownarrow\!*),\\
& & \quad (a_1\!\updownarrow \dots a_k\!\updownarrow,\ b_1\!\updownarrow \dots b_n\!\updownarrow\!\#c_1\!\uparrow \dots c_m\!\uparrow) \ \ |\ \ k < n,\\
& & \qquad\qquad (a_1\dots a_k\,,\ b_1\dots b_n) \in A \text{ and } c_1\dots c_m \in D_\ell\} \ \cup\\
& & \{\ (a_1\!\updownarrow \dots a_k\!\updownarrow\!*,\ b_1\!\updownarrow \dots b_n\!\updownarrow),\\
& & \quad (a_1\!\updownarrow \dots a_k\!\updownarrow\!\#c_1\!\uparrow \dots c_m\!\uparrow,\ b_1\!\updownarrow \dots b_n\!\updownarrow) \ \ |\ \ k > n,\\
& & \qquad\qquad (a_1\dots a_k\,,\ b_1\dots b_n) \in A \text{ and } c_1\dots c_m \in D_u\} \ \cup\\
& & \{\ (a_1\!\updownarrow \dots a_k\!\updownarrow\!\#c_1\!\uparrow \dots c_m\!\uparrow d_1\!\downarrow \dots d_l\!\downarrow\!*,\ b_1\!\updownarrow \dots b_k\!\updownarrow) \ \ |\ \ k > n,\\
& & \qquad (a_1\dots a_k\,,\ b_1\dots b_n) \in A \text{ and } c_1\dots c_m\,,\ d_1\dots d_l \in D_u\} \ \cup\\
& & \{\ (a_1\!\updownarrow \dots a_k\!\updownarrow\!\#c_1\!\uparrow \dots c_m\!\uparrow,\ b_1\!\updownarrow \dots b_k\!\updownarrow\!\#d_1\!\uparrow \dots d_n\!\uparrow) \ \ |\ k = n,\\
& & \quad (a_1\dots a_k\,,\ b_1\dots b_k) \in A\,,\ c_1\dots c_m \in D_u \text{ and } d_1\dots d_n \in D_\ell\}
\end{array}
$$

The coding h is defined by $h(a\uparrow) = h(a\downarrow) = h(\overline{a}\uparrow) = h(\overline{a}\downarrow) = h(a\downarrow*) = h(\overline{a}\downarrow*) = h(a\updownarrow) = h(\overline{a}\updownarrow) = h(a\updownarrow\#) = h(\overline{a}\updownarrow\#) = a$ for each $a \in \Sigma$.

Note that ρ is defined such that the arrows $\uparrow, \downarrow, \updownarrow$ and $\updownarrow\#$ must match the decoration of the corresponding letter in the other strand, whereas bars ($\overline{a}$) and stars ($*$) must *not* match.

Correctness. First, as in the proof of Lemma 9.1, it is clear that all computations of γ' are primitive.

Second, we show that $L_f(\gamma) \subseteq h(L_f(\gamma'))$. Every computation of γ can also be written as $(u_1 \dots u_k, v_1 \dots v_k)$, with the same properties as in the proof of Lemma 9.1. Following the same reasoning as in that proof, and in view of the definition of γ' and the explanations directly before this lemma, it is clear that for each computation of γ an equivalent (modulo h) primitive computation of

γ' can be given. The correspondence between such a pair of computations in γ and γ' is represented schematically for three sample computations in Figures 9.1 and 9.2, where Figure 9.1(a) corresponds to Figure 9.2(a) and so on.

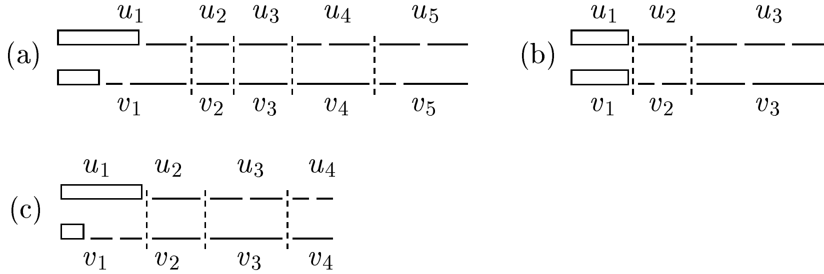


Figure 9.1: Three computations in γ

Here a vertical dashed line indicates an intermediate blunt end (a real one in Figure 9.1 and a 'prevented' one in Figure 9.2), the two boxes in every computation represent the two parts of an axiom, and each solid horizontal line is a sticker. The black dots in Figure 9.2 indicate the positions where two 'normal' stickers are concatenated to form a double-sticker, and an arrow placed on a line or a box means that every letter in that sticker or axiom (until the next black dot) has that arrow with it. We did not draw the $*$ and # symbols.

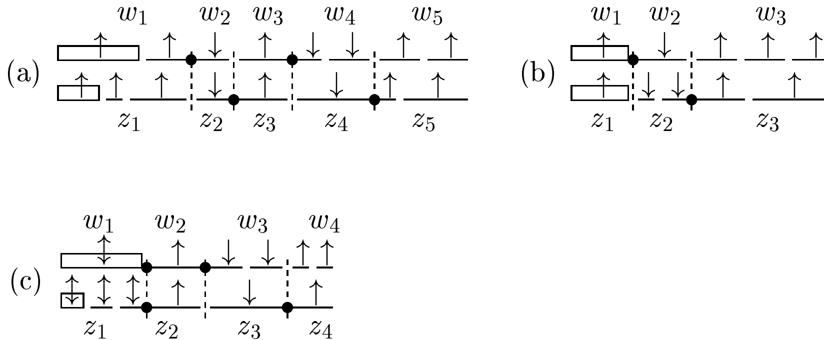


Figure 9.2: Three computations in γ'

It should be clear that the difference between the number of upper stickers and the number of lower stickers used in the computation of γ is the same as in the corresponding computation of γ'. As explained, this is due to the alternation of $\uparrow$ and $\downarrow$ in the 'segments', forcing the concatenation of stickers to occur an equal number of times in the upper and lower strand; the mark $*$ prohibits an unmatched concatenation in the upper strand.

Third, for the same reason as in the proof of Lemma 9.1, it is clear that $h(L(\gamma')) \subseteq L(\gamma)$.

Finally, we show that for each fair computation (x, y) of γ' there is a fair computation $(h(x), h(y))$ of γ. Define $\Sigma_{\uparrow} = \{a\uparrow, \overline{a}\uparrow \mid a \in \Sigma\}$, $\Sigma_{\downarrow} = \{a\downarrow, \overline{a}\downarrow, a\downarrow *, \overline{a}\downarrow * \mid a \in \Sigma\}$ and $\Sigma_{\updownarrow} = \{a\updownarrow, \overline{a}\updownarrow, a\updownarrow\#, \overline{a}\updownarrow\# \mid a \in \Sigma\}$. A careful look at the definition of γ' reveals that each computation of γ' can also be written as $(w_1 \dots w_n, z_1 \dots z_n)$, for a certain $n \geq 1$, where either $w_i, z_i \in \Sigma_{\uparrow}^+$ for i odd and $w_i, z_i \in \Sigma_{\downarrow}^+$ for i even, or $w_1, z_1 \in \Sigma_{\updownarrow}^+$, $w_i, z_i \in \Sigma_{\uparrow}^+$ for i even and $w_i, z_i \in \Sigma_{\downarrow}^+$ for i odd. Since every sticker that ends with a symbol from $\Sigma_{\downarrow}$ is marked with a $*$ in this last symbol, and since ρ' defines that two symbols that are both marked with $*$ are not complementary, it follows that $w_1, z_1 \in \Sigma_{\uparrow}^+$ if and only if n is odd, and $w_1, z_1 \in \Sigma_{\updownarrow}^+$ if and only if n is even.

For i odd, the only way to cross the border between $(w_i, z_i) \in \Sigma_{\uparrow}^+ \times \Sigma_{\uparrow}^+$ and $(w_{i+1}, z_{i+1}) \in \Sigma_{\downarrow}^+ \times \Sigma_{\downarrow}^+$ is to use a double-sticker in the upper strand. Similarly, for i even, the only way to change from $(w_i, z_i) \in \Sigma_{\downarrow}^+ \times \Sigma_{\downarrow}^+$ to $(w_{i+1}, z_{i+1}) \in \Sigma_{\uparrow}^+ \times \Sigma_{\uparrow}^+$ is to use a double-sticker in the lower strand. Note that there are no lower double-stickers in $\Sigma_{\uparrow}^+ \cdot \Sigma_{\downarrow}^+$ and no upper double-stickers in $\Sigma_{\downarrow}^+ \cdot \Sigma_{\uparrow}^+$. Moreover, double-stickers can only be used on such a border, because every double-sticker contains at least two different kinds of arrow, whereas every 'blunt segment' (w_i, z_i) contains only one kind of arrow.

Furthermore, no computation can consist of only symbols from $\Sigma_{\updownarrow}$, and the only way to finish a computation that starts with an axiom beginning with symbols from $\Sigma_{\updownarrow}$ is to use in one of the two strands something that contains a # (i.e., a double-sticker or an axiom with a sticker attached to at least one of the two components). From the definition of ρ' it follows that symbols containing # are complementary only to themselves, which guarantees that at that position a double-sticker or an axiom concatenated with a sticker is used in both strands.

In short, the symbols $\uparrow, \downarrow, *$ and # together guarantee that in each computation of γ' the number of concatenations (the black dots in Figure 9.2) used in the upper strand equals the number of concatenations used in the lower strand. Hence when in a fair computation of γ' these concatenations are 'detached' again we have a fair computation of γ that generates the same string, modulo h. □

9.2 Primitive computations

In the previous section we gave a construction to translate a sticker system γ into an equivalent (modulo a coding) sticker system γ' that can do only primitive computations. When we only wish to satisfy the weaker requirement that $L(\gamma)$ equals $L_p(\gamma')$, i.e., not *all* computations of γ' have to be primitive, then we can use a simple variant of the construction in the proof of Lemma 9.1: just

remove the markings and the coding. The result is described in the following theorem.

Theorem 9.4 $\mathsf{SL} \subseteq \mathsf{SL}_p$

Proof. *Construction.* Let $\gamma = (\Sigma, \rho, D_u, D_\ell, A)$ be a sticker system. We construct a sticker system $\gamma' = (\Sigma, \rho, D'_u, D'_\ell, A')$ with $L_p(\gamma') = L(\gamma)$ as follows.

$$\begin{aligned} D'_u &= D_u \cup \{xy \mid x, y \in D_u\} \\ D'_\ell &= D_\ell \cup \{xy \mid x, y \in D_\ell\} \\ A' &= A \cup \{(x_0 x, y_0) \mid (x_0, y_0) \in A \text{ and } x \in D_u\} \end{aligned}$$

Correctness. Clearly, for each computation of γ, the new sticker system γ' can do both exactly the same computation and a primitive computation that generates the same string. Now by the proof of Lemma 9.1, if we remove from that proof all reference to the markings and the coding, we have $L(\gamma) = L_p(\gamma')$. □

Hence this yields a true primitive normal form for sticker languages (as opposed to the primitive 'normal form' from the previous section, where a coding was needed): for each sticker language there is a sticker system that needs only primitive computations to generate it.

Now let us return to Example 7.1, where we show that $ba^*b \in \mathsf{SL}_p$. Since it is known that $ba^*b \notin \mathsf{SL}$ (see page 82), we can refine the inclusion in Theorem 9.4 to a proper inclusion. Indeed, the reason that ba^*b is not in SL is that computations in general may be non-primitive and therefore can also derive words from ba^*ba^+ in any sticker system generating all words from ba^*b ([PR98, Theorem 10]). This already suggests that it is possible to generate ba^*b when only the primitive computations are taken into account.

Corollary 9.5 $\mathsf{SL} \subset \mathsf{SL}_p$

Hence we have $\mathsf{SL} \subset \mathsf{SL}_p \subseteq \mathsf{REG}$. We would now like to know whether the latter inclusion is proper or not. A candidate language to show that it is proper could be a^*ba^*, because it seems unlikely that any computation of any sticker system can 'know' whether it has already generated the b or not. However, the following example shows that this problem can be overcome by coding the 'state' into the length of the sticky ends. Since $a^*ba^* \notin \mathsf{SL}$, the example makes essential use of the selectiveness of primitive computations.

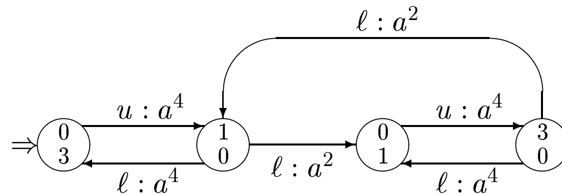
Example 9.1 We show that $a^*ba^* \in \mathsf{SL}_p$. The idea is that we need at least one upper and one lower sticker that both consist of only a's, plus one upper and one lower sticker that both contain one b and possibly some a's before or after it. Moreover, when the latter two stickers appear together in a computation, the

b's have to match (we use the identity on $\{a, b\}$ as complementarity relation), and since we only consider primitive computations this must leave a non-empty overhang left of the b and a non-empty overhang right of the b (assuming that the computation is long enough). To get the correct number of b's in the resulting string, we have to be able to distinguish between the a's before the b and the a's after the b. Since in those parts of the string we can only use stickers consisting entirely of a's, the only way to do this is by ensuring that the overhangs occurring before the b are different from the overhangs occurring after the b.

Let γ be the following sticker system:

$$\begin{aligned}
\Sigma &= \{\, a \,,\, b \,\} \\
\rho &= id \\
D_u &= \{\, a^4 \,,\, b \,,\, aba \,,\, ba^2 \,,\, aba^3 \,\} \\
D_\ell &= \{\, a^4 \,,\, ab \,,\, ba \,,\, aba^2 \,,\, ba^3 \,,\, a^2 \,\} \\
A &= \{\, (\lambda, a^3) \,,\, (a, a^4) \,,\, (a^2, a^5) \,,\, (a^3, a^6) \,\} \;\cup \\
&\quad\; \{\, (a^i b, a^i b) \,,\, (a^i ba, a^i ba) \,,\, (a^i ba^2, a^i b), \\
&\qquad (a^i ba^3, a^i ba) \,,\, (a^i ba^4, a^i ba^2) \;\mid\; 0 \le i \le 4 \,\}
\end{aligned}$$

Observe that the axioms that consist of only a's all have a sticky end of length 3 in the lower strand. Now, when we add stickers containing only a's (i.e., a^4 in the upper strand and a^4, a^2 in the lower strand) to axioms of this kind, there are only four sticky ends that can occur, provided that we add the stickers in such a way that the current sticky end is not prolonged. The relations between these stickers and overhangs are depicted in the finite automaton below, in which we have indicated, e.g., an overhang of 3 a's in the lower strand by the state $\genfrac{}{}{0pt}{}{0}{3}$. When the upper sticker a^4 is added to a computation that ends with such an overhang, a new overhang of one a in the upper strand is created, denoted by a transition from $\genfrac{}{}{0pt}{}{0}{3}$ to $\genfrac{}{}{0pt}{}{1}{0}$ labelled by '$u : a^4$'. In the new state $\genfrac{}{}{0pt}{}{1}{0}$ it is possible to add lower sticker a^4 (denoted by a transition labelled by $\ell : a^4$) or a^2 ($\ell : a^2$), and so on.

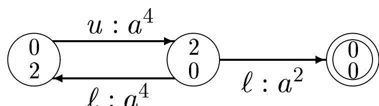


Since the above automaton does not have a state $\genfrac{}{}{0pt}{}{0}{0}$, it is clear that computations without a b are never complete.

Note that the only matching pairs consisting of an upper and a lower sticker, both containing a b, that can be used in a primitive computation of γ are the following (the pairs are indicated by writing the upper sticker on top of the lower sticker, with the b's matching):

b	b	aba	aba	baa	baa	$abaaa$	$abaaa$
ab	$abaa$	ba	$baaa$	ab	$abaa$	ba	$baaa$

Indeed, in each of these pairs the overhangs before the b (length 1) differ from those after the b (length 0 or 2), and each of these pairs can be attached to a computation ending in state $\genfrac{}{}{0pt}{}{0}{1}$ or $\genfrac{}{}{0pt}{}{1}{0}$ in the automaton depicted above. Moreover, after using any such pair we can finish the computation using a^4 in the upper and a^4, a^2 in the lower strand. The sticky ends that are possible when we add stickers to the overhangs $\genfrac{}{}{0pt}{}{0}{2}$ and $\genfrac{}{}{0pt}{}{2}{0}$ are given in the automaton below (since we consider only primitive computations, we have to stop when reaching the overhang $\genfrac{}{}{0pt}{}{0}{0}$).



Clearly, when we have reached one of these three overhangs, it is not possible to prolong the computation by using stickers containing b's. Note that the axioms that contain a b all leave overhangs of length 0 or 2 as well. Consequently, our sticker system ensures that every string in its primitive language is of the form a^*ba^*.

Suppose that we use one of the axioms containing only a's. When in the upper strand the stickers b or ba^2 are used, then there are $4j + i$ a's in front of the b, for $j \geq 1$ and $0 \leq i \leq 3$, and after the b there are either $4k$ a's or $4k + 2$ a's, for $k \geq 0$, respectively. Similarly, when in the upper strand the stickers aba or aba^3 are used, then there are $4j + i$ a's in front of the b, for $j \geq 1$ and $1 \leq i \leq 4$, and after the b there are either $4k + 1$ a's or $4k + 3$ a's, for $k \geq 0$, respectively. Thus, when using only the axioms consisting of a's, our sticker system can generate all strings of the form a^iba^j for $i \geq 5$ and $j \geq 0$, using only primitive computations. Since primitive computations starting with the other axioms in A yield all strings of the form a^iba^j for $0 \leq i \leq 4$ and $j \geq 0$, we have $a^*ba^* \subseteq L_p(\gamma)$. □

A logical next step would be to investigate, e.g., whether $a^*ba^*ba^*$ is also in SL_p, since for a^*ba^* we needed two disjoint groups of overhangs, namely $\{ \genfrac{}{}{0pt}{}{0}{3}, \genfrac{}{}{0pt}{}{1}{0}, \genfrac{}{}{0pt}{}{0}{1}, \genfrac{}{}{0pt}{}{3}{0} \}$ and $\{ \genfrac{}{}{0pt}{}{0}{2}, \genfrac{}{}{0pt}{}{2}{0}, \genfrac{}{}{0pt}{}{0}{0} \}$, but for $a^*ba^*ba^*$ we would need three. Although it is possible to create three disjoint groups of overhangs by using, for instance, a^6 as an upper

sticker and a^6 and a^3 as lower stickers, which gives the disjoint sets $\{ {}^{0}_{1}, {}^{5}_{0}, {}^{2}_{0}, {}^{0}_{4} \}$, $\{ {}^{0}_{2}, {}^{4}_{0}, {}^{1}_{0}, {}^{0}_{5} \}$ and $\{ {}^{0}_{3}, {}^{3}_{0}, {}^{0}_{0} \}$, the question whether there exists a sticker system that generates $a^*ba^*ba^*$ as its primitive language remains unanswered.

9.3 Fair computations

Rather surprisingly, it is also possible to generate each sticker language with only fair computations. The idea behind this is that when a computation alternatingly uses an upper sticker leaving a sticky end in the upper strand and a lower sticker leaving a sticky end in the lower strand, then the difference between the numbers of upper and lower stickers used can be at most one, i.e., the computation is either fair or almost fair. Of course, in general it is not the case that sticky ends occur alternatively in the upper and lower strand. For instance, it can happen that the only lower sticker that matches a sticky end in the upper strand is not long enough to match the entire overhang. In such a case we use a similar technique as when making a computation primitive: we concatenate as many lower stickers as necessary to create either a sticky end in the lower strand or a complete computation. Since there is only a finite number of axioms and a finite number of stickers, it can be determined beforehand how many stickers should be concatenated in the worst case.

Theorem 9.6 $\mathsf{SL} \subseteq \mathsf{SL}_f$

Proof. *Construction.* Let $\gamma = (\Sigma, \rho, D_u, D_\ell, A)$ be a sticker system. We construct a sticker system $\gamma' = (\Sigma, \rho, D'_u, D'_\ell, A')$ with $L_f(\gamma') = L(\gamma)$ as follows.

Let $d = \max\{d_a, d_s\}$, with $d_a = \max\{|x_0| - |y_0|, |y_0| - |x_0| \mid (x_0, y_0) \in A\}$ and $d_s = \max\{|x| \mid x \in D_u \cup D_\ell\}$. Now define

$$\begin{aligned}
D'_u &= \{x_1 \dots x_k \mid x_1, \dots, x_k \in D_u \text{ for some } k \geq 1 \text{ such that} \\
&\qquad\qquad |x_1 \dots x_k| \leq d + \max\{\, |x| \mid x \in D_u\}\} \\
D'_\ell &= \{x_1 \dots x_k \mid x_1, \dots, x_k \in D_\ell \text{ for some } k \geq 1 \text{ such that} \\
&\qquad\qquad |x_1 \dots x_k| \leq d + \max\{\, |x| \mid x \in D_\ell\}\} \\
A' &= A \;\cup \\
&\quad \{(x_0 x', y_0), (x_0, y_0 y'), (x_0 x, y_0 y') \mid (x_0, y_0) \in A, x \in D_u, \\
&\qquad\qquad x' \in D'_u \text{ and } y' \in D'_\ell\}
\end{aligned}$$

Correctness. It is obvious that $L(\gamma') \subseteq L(\gamma)$, hence $L_f(\gamma') \subseteq L(\gamma)$.

Let $(x_0 x_1 \dots x_m, y_0 y_1 \dots y_n)$ be a complete computation of γ. If $|x_0| > |y_0|$, then $|x_0| - |y_0| \leq d$. Let j be the smallest index $(1 \leq j \leq n)$ such that $|y_1 \dots y_j| > |x_0| - |y_0|$ (or such that $|y_1 \dots y_j| = |x_0| - |y_0|$ if $m = 0$), then $|y_1 \dots y_j| \leq d + \max\{|y| \mid y \in D_\ell\}$, hence $y_1 \dots y_j \in D'_\ell$.

Now $0 \leq |y_0 y_1 \dots y_j| - |x_0| \leq d$. Assume that $0 < |y_0 y_1 \dots y_j| - |x_0|$ (otherwise we proceed with the next paragraph). Let i be the smallest index ($1 \leq i \leq m$) such that $|x_1 \dots x_i| > |y_0 y_1 \dots y_j| - |x_0|$ (or such that $|x_1 \dots x_i| = |y_0 y_1 \dots y_j| - |x_0|$ if $j = n$), then $|x_1 \dots x_i| \leq d + \max \{|x| \mid x \in D_u\}$, thus $x_1 \dots x_i \in D'_u$. And so on, until $i = m$ and $j = n$.

Clearly this way of alternatingly adding an upper and a lower sticker to the computation, such that the current sticky end is at least completely matched, guarantees that in the end either we used one lower sticker 'too many' (if we end with a lower sticker) or the computation is fair (if we end with an upper sticker). In the former situation we make the computation of γ' fair by using the axiom $(x_0, y_0 y_1 \dots y_j) \in A'$ instead of $(x_0, y_0) \in A'$.

The case where $|y_0| \geq |x_0|$ can be treated analogously (if $|x_0| = |y_0|$, then use the axiom $(x_0 x_1, y_0) \in A'$ and find the smallest index j ($1 \leq j \leq n$) such that $|y_1 \dots y_j| \geq |x_1|$ etcetera).

Consequently also $L(\gamma) \subseteq L_f(\gamma')$. □

Note that the computations in γ' constructed in the previous proof are not only fair, but also primitive.

Corollary 9.7 $\mathsf{SL} \subseteq \mathsf{SL}_{pf}$

We know from Lemma 8.2 that $ba^*b \notin \mathsf{SL}_f$, and from Example 7.1 that $ba^*b \in \mathsf{SL}_p$. Hence $\mathsf{SL}_p \not\subseteq \mathsf{SL}_f$.

Moreover, Example 7.2 shows that the non-regular language $K = \{x \in \{aa, bb\}^* \mid \#_a(x) = \#_b(x)\}$ is in SL_f. Obviously K cannot be in SL_p, hence $\mathsf{SL}_f \not\subseteq \mathsf{SL}_p$.

Lemma 9.8 SL_p *and* SL_f *are incomparable.*

Furthermore, because of the language K we can refine the inclusion $\mathsf{SL} \subseteq \mathsf{SL}_f$ to a proper inclusion.

Corollary 9.9 $\mathsf{SL} \subset \mathsf{SL}_f$

9.4 Primitive fair computations

Clearly the primitive fair sticker languages combine some properties of primitive and fair sticker languages: when only primitive computations are concerned, one has some control over when a computation ends, while fairness allows one to count and thus to generate non-regular languages. Therefore, the following two examples are not surprising.

Example 9.2 The regular language ba^*b, that is in SL_p but not in SL nor in SL_f, is in SL_{pf}: consider the sticker system

$$\gamma = (\{a,b\}, id, \{aa, ab\}, \{aa, b\}, \{(ba, baa), (b, ba), (bb, bb)\}),$$

that differs only from the sticker system given in Example 7.1 in the axiom (ba, baa). Primitive fair computations that start with (b, ba) always generate an odd number of a's:

$$\begin{matrix} [b\rangle\langle a\ b\rangle \\ [b\ a\rangle\langle b\rangle \end{matrix} \qquad \begin{matrix} [b\rangle\langle aa\rangle\,\langle a\ b\rangle \\ [b\ a\rangle\langle\, aa\rangle\langle b\rangle \end{matrix} \qquad \text{etcetera,}$$

while the primitive fair computations of γ that start with (ba, baa) always generate an even number of a's:

$$\begin{matrix} [ba\rangle\langle a\ b\rangle \\ [ba\ a\rangle\langle b\rangle \end{matrix} \qquad \begin{matrix} [ba\rangle\langle aa\rangle\,\langle a\ b\rangle \\ [ba\ a\rangle\langle\, aa\rangle\langle b\rangle \end{matrix} \qquad \text{etcetera.}$$

Clearly the three axioms of γ ensure in this way that $L_{pf}(\gamma) = ba^*b$. □

Example 9.3 Let $\gamma = (\{a,b,c\}, id, D_u, D_\ell, A)$ be the sticker system defined by

$$\begin{aligned} D_u &= \{bbc, a, ac\} \\ D_\ell &= \{caa, cb, b\} \\ A &= \{(c, \lambda)\} \end{aligned}$$

There are only two ways for a (long enough) primitive computation to start:

$$\begin{matrix} [\ \ c\rangle\langle a\rangle\langle a\ c\rangle \\ [\rangle\langle c\ \ a\ \ a\rangle \end{matrix} \qquad \text{and} \qquad \begin{matrix} [\ \ c\rangle\langle b\ \ b\ c\rangle \\ [\rangle\langle c\ \ b\rangle\langle b\rangle \end{matrix}$$

Both these ways leave a sticky end consisting of one c in the upper strand, which is the same as the sticky end left by the axiom. Hence each primitive computation of γ is composed of two 'building blocks', one of which consists of the upper stickers a and ac and the lower sticker caa, and the other of upper sticker bbc and lower stickers cb and b. The only way to end a computation is with such a block of a's (use upper sticker a instead of ac).

Furthermore, note that each block of a's as described above uses two upper stickers and one lower sticker, whereas each block of b's uses one upper sticker and two lower stickers. Therefore, primitive computations of γ are only fair when an equal amount of blocks of a's and blocks of b's is used. Consequently, $L_{pf}(\gamma) = \{w \in \{caa, cbb\}^* \cdot caa \mid \#_a(w) = \#_b(w)\}$. □

Because of these examples we now also have

Lemma 9.10 $\mathsf{SL} \subset \mathsf{SL}_{pf}$, $\mathsf{SL}_{pf} - \mathsf{SL}_f \neq \varnothing$ *and* $\mathsf{SL}_{pf} - \mathsf{SL}_p \neq \varnothing$.

In view of the previous results on primitive and fair sticker languages, i.e., (1) each fair sticker language is a BCA-language because a BCA can keep track of both the sticky ends and the numbers of stickers used in a computation, and (2) each primitive sticker language is regular since simulation of non-primitive computations by a DFA can be prevented by removing all transitions that start in the final state, the following lemma is not very surprising.

Lemma 9.11 $\mathsf{SL}_{pf} \subseteq \mathsf{1BCA}$

Proof. In the proof of Theorem 8.1, where we show that each fair sticker language is a BCA-language, we use BCA's $\mathcal{B}_{x_0,y_0}$ with $L(\mathcal{B}_{x_0,y_0}) = L_f(\gamma_{x_0,y_0})$, where $\gamma = (\Sigma, \rho, D_u, D_\ell, A)$ is the sticker system under consideration, (x_0, y_0) is in A and $\gamma_{x_0,y_0} = (\Sigma, \rho, D_u, D_\ell, \{(x_0, y_0)\})$.

Clearly, if we remove all transitions starting in the final state of $\mathcal{B}_{x_0,y_0}$, we have a BCA $\mathcal{B}'_{x_0,y_0}$ with $L(\mathcal{B}'_{x_0,y_0}) = L_{pf}(\gamma_{x_0,y_0})$. Consequently $L_{pf}(\gamma) = \bigcup_{(x_0,y_0)\in A} L(\mathcal{B}'_{x_0,y_0})$ is a BCA-language. □

Combining $\mathsf{1BCA} \subseteq \mathsf{COD}(\mathsf{SL}_{pf})$ (Lemma 9.2), the previous lemma and the fact that 1BCA is closed under codings now gives the following result.

Corollary 9.12 $\mathsf{SL}_{pf} \subseteq \mathsf{1BCA} = \mathsf{COD}(\mathsf{SL}_{pf})$

Concerning SL_p and SL_f, the problem of finding constructions to change primitive or fair computations into primitive fair computations is still open (i.e., it is open whether $\mathsf{SL}_p \subseteq \mathsf{SL}_{pf}$ and $\mathsf{SL}_f \subseteq \mathsf{SL}_{pf}$). The problem here is that one should avoid that non-primitive or non-fair computations, respectively, become primitive fair.

9.5 Summary

We recall the most interesting results of this chapter:

1. $\mathsf{SL} \subset \mathsf{SL}_p$, $\mathsf{SL} \subset \mathsf{SL}_f$, $\mathsf{SL} \subset \mathsf{SL}_{pf}$,
2. SL_p and SL_f are incomparable,
3. $\mathsf{SL}_{pf} \subseteq \mathsf{1BCA}$,
4. $\mathsf{SL}_{pf} - \mathsf{SL}_f \neq \varnothing$ and $\mathsf{SL}_{pf} - \mathsf{SL}_p \neq \varnothing$,
5. $a^*ba^* \in \mathsf{SL}_p$.

There were also some problems that we could not solve: is $a^*ba^*ba^* \in \mathsf{SL}_p$, is $\mathsf{SL}_p \subseteq \mathsf{SL}_{pf}$, and is $\mathsf{SL}_f \subseteq \mathsf{SL}_{pf}$?

Part III

Forbidding and enforcing

Chapter 10

Definitions, examples and research topics

We describe a model of molecular computing that is based on boundary conditions, that specify the boundaries within which a system may evolve. More specifically, we consider forbidding conditions, that prevent certain things from occurring, and enforcing conditions, that, under the right conditions, cause certain things to occur.

10.1 Forbidding

Forbidding conditions describe the situation where a system will 'die' whenever a certain group of components (parts of molecules) is present in this system. We formalize these conditions by forbidding sets, as follows.

Definition 10.1 A *forbidding set* is a (possibly infinite) family of finite languages over some alphabet Σ; these finite languages are called *forbidders*.

A language $K \subseteq \Sigma^*$ is *consistent* with a forbidder $F \subseteq \Sigma^*$, denoted $K \underline{\text{con}}\ F$, if $F \not\subseteq \underline{\text{sub}}\,(K)$. A language K is consistent with a forbidding set $\mathcal{F}$, denoted $K \underline{\text{con}}\ \mathcal{F}$, if K is consistent with every forbidder in $\mathcal{F}$. □

Example 10.1 Consider the forbidding set $\mathcal{F} = \{\{ab, ba\}, \{aa, bb\}\}$ and a language $K \subseteq \{a, b\}^*$. Then it is easily seen that $K \underline{\text{con}}\ \mathcal{F}$ if and only if $K \subseteq K_i$ for some $i \in \{1, 2, 3, 4\}$, where $K_1 = ab^* \cup b^*$, $K_2 = a^*b \cup a^*$, $K_3 = ba^* \cup a^*$ and $K_4 = b^*a \cup b^*$. □

For a forbidding set $\mathcal{F}$ we define the family of ($\mathcal{F}$-)consistent languages

$$\mathcal{L}(\mathcal{F}) = \{K \mid K \underline{\text{con}}\ \mathcal{F}\}$$

and the family of finite consistent languages

$$\mathcal{L}_{\text{fin}}(\mathcal{F}) = \{K \mid K \text{ is finite and } K \underline{\text{con}}\ \mathcal{F}\}.$$

Note that $\varnothing \in \mathcal{L}_{\text{fin}}(\mathcal{F})$ for every $\mathcal{F}$.

We say that two forbidding sets $\mathcal{F}_1$ and $\mathcal{F}_2$ are equivalent, denoted $\mathcal{F}_1 \sim \mathcal{F}_2$, if $\mathcal{L}(\mathcal{F}_1) = \mathcal{L}(\mathcal{F}_2)$.

Example 10.2 Suppose that $\{\lambda\}$ is a forbidder, contained in a forbidding set $\mathcal{F}$. Then $K \underline{\text{con}}\ \{\lambda\}$ if and only if $\{\lambda\} \not\subseteq \underline{\text{sub}}(K)$ if and only if $K = \varnothing$. Since a language is consistent with $\mathcal{F}$ only if it is consistent with $\{\lambda\}$, we have $\mathcal{L}(\mathcal{F}) = \{\varnothing\}$. □

Example 10.3 Let $\{\lambda, a, bb\}$ be a forbidder and let $\Sigma = \{a, b, c\}$ be an alphabet. Then $K \underline{\text{con}}\ \{\lambda, a, bb\}$ if and only if $\{\lambda, a, bb\} \not\subseteq \underline{\text{sub}}(K)$, which is equivalent to $\{a, bb\} \not\subseteq \underline{\text{sub}}(K)$ since $\lambda \in \underline{\text{sub}}(w)$ for each $w \in \Sigma^*$. Therefore $\mathcal{L}(\{\{\lambda, a, bb\}\}) = \mathcal{L}(\{\{a, bb\}\}) = \{K \mid K \subseteq \{b, c\}^*\} \cup \{K \subseteq \Sigma^* \mid bb \notin \underline{\text{sub}}(K)\}$. □

Example 10.4 Let $\mathcal{F}$ be the forbidding set $\{\{ba^ib\} \mid i \geq 1\}$, and let $K \subseteq \{a, b\}^*$ be a language. Then $K \underline{\text{con}}\ \mathcal{F}$ if and only if $ba^ib \notin \underline{\text{sub}}(K)$ for all $i \geq 1$, i.e., $K \underline{\text{con}}\ \mathcal{F}$ if and only if $K \subseteq \{w \in \{a, b\}^* \mid ba^ib \notin \underline{\text{sub}}(w)$ for all $i \geq 1\}$. Since the latter language equals $a^*b^*a^*$, when we restrict ourselves to the alphabet $\{a, b\}$ we obtain $\mathcal{L}(\mathcal{F}) = \{K \mid K \subseteq a^*b^*a^*\}$. □

From [ER, EH$^+$00] we recall three basic properties of forbidding.

Proposition 10.1 *Let $\mathcal{F}$ be a forbidding set and K a language.*
(1) $K \underline{\text{con}}\ \mathcal{F}$ implies $\underline{\text{sub}}(K) \underline{\text{con}}\ \mathcal{F}$
(2) If $K' \subseteq K$ and $K \underline{\text{con}}\ \mathcal{F}$, then $K' \underline{\text{con}}\ \mathcal{F}$
(3) If $K_1, K_2, \ldots$ is an ascending sequence of languages with $K_i \underline{\text{con}}\ \mathcal{F}$ for all $i \geq 1$, then $(\bigcup_{i\geq 1} K_i)\ \underline{\text{con}}\ \mathcal{F}$

The third property above follows from the definition of forbidders as finite sets.

Some other simple properties of forbidding are the following. From the definitions it is immediately clear that, for two forbidding sets $\mathcal{F}$ and $\mathcal{F}'$ with $\mathcal{F}' \subseteq \mathcal{F}$, we have $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}')$.

Note that from property (2) it follows that $K \underline{\text{con}}\ \mathcal{F}$ implies $(K \cap K') \underline{\text{con}}\ \mathcal{F}$, for any K'. Hence it is also clear that $\mathcal{L}(\mathcal{F})$ is closed under intersection: if $K \underline{\text{con}}\ \mathcal{F}$ and $K' \underline{\text{con}}\ \mathcal{F}$, then $(K \cap K') \underline{\text{con}}\ \mathcal{F}$.

Furthermore, $\mathcal{L}(\mathcal{F})$ is not closed under union. Take for instance $\mathcal{F} = \{\{a, b\}\}$, $K = \{a\}$ and $K' = \{b\}$, then obviously $K \underline{\text{con}}\ \mathcal{F}$ and $K' \underline{\text{con}}\ \mathcal{F}$, whereas $K \cup K'$ is not consistent with $\mathcal{F}$.

10.2 Enforcing

In this section we formalize enforcing conditions, where the presence of a certain group of molecules causes the presence of at least one member from another group of molecules.

10.2.1 Definitions and basic properties

Definition 10.2 An *enforcing set* is a (possibly infinite) family of ordered pairs (X, Y), where X and Y are finite languages over some alphabet Σ, with $Y \neq \varnothing$; such a pair (X, Y) is called an *enforcer*.

A language $K \subseteq \Sigma^*$ *satisfies* an enforcer (X, Y) with $X, Y \subseteq \Sigma^*$, denoted $K\, \underline{\mathrm{sat}}\, (X, Y)$, if $X \subseteq K$ implies $Y \cap K \neq \varnothing$. A language K satisfies an enforcing set $\mathcal{E}$, denoted $K\, \underline{\mathrm{sat}}\, \mathcal{E}$, if K satisfies every enforcer in $\mathcal{E}$. □

Let $\mathcal{E}$ be an enforcing set. An enforcer $(X, Y) \in \mathcal{E}$ is *applicable* to a language K if $X \subseteq K$. If (X, Y) is applicable to K, but $Y \cap K = \varnothing$, then (X, Y) is a K*-violator*.

Example 10.5 The family $\mathcal{E} = \{(\{u, v\}, \{uv, vu\}) \mid u, v \in \Sigma^+\}$ is an enforcing set. If $K \subseteq \Sigma^+$ satisfies $\mathcal{E}$, then K is closed under 'weak catenation': for any two words $u, v \in K$ at least one of the words uv, vu is in K. Note that there are infinitely many languages satisfying $\mathcal{E}$, each resulting from a different 'implementation' of the weak catenation. □

Note the non-deterministic nature of enforcers (X, Y), that is caused by allowing Y to include more than one element and by requiring that, given an enforcer (X, Y) and a language K, if the set X of premises is included in K, then *at least one* element of the set Y of consequences will be included in K.

Example 10.6 Let $\mathcal{E}$ be the enforcing set $\{(\varnothing, \{b^n\}) \mid n \text{ is even}\}$. We refer to each enforcer $(\varnothing, \{b^n\})$ as a 'brute' enforcer, because its premise, the empty set, is included in every language. Thus if K satisfies $\mathcal{E}$, then K must contain the language $\{b^n \mid n \text{ is even}\}$. □

For an enforcing set $\mathcal{E}$,

$$\mathcal{L}(\mathcal{E}) = \{K \mid K\, \underline{\mathrm{sat}}\, \mathcal{E}\}$$

is the family of ($\mathcal{E}$-)satisfying languages. Similarly, the family of finite satisfying languages is defined by

$$\mathcal{L}_{\mathrm{fin}}(\mathcal{E}) = \{K \mid K \text{ is finite and } K\, \underline{\mathrm{sat}}\, \mathcal{E}\}.$$

We say that two enforcing sets $\mathcal{E}_1$ and $\mathcal{E}_2$ are equivalent, denoted $\mathcal{E}_1 \sim \mathcal{E}_2$, if $\mathcal{L}(\mathcal{E}_1) = \mathcal{L}(\mathcal{E}_2)$.

It is obvious that $\mathcal{L}(\mathcal{E}) \subseteq \mathcal{L}(\mathcal{E}')$ for any two enforcing sets $\mathcal{E}$ and $\mathcal{E}'$ with $\mathcal{E}' \subseteq \mathcal{E}$.

Considering closure under intersection and union, the following two small examples show that $\mathcal{L}(\mathcal{E})$ is closed under neither of the two operations. Let $\mathcal{E} = \{(\{a\}, \{b, c\})\}$, and let $K = \{a, b\}$ and $K' = \{a, c\}$, then obviously both K and K' satisfy $\mathcal{E}$, but $K \cap K' = \{a\}$ does not. On the other hand, if $\mathcal{E} = \{(\{a, b\}, \{c\})\}$, $K = \{a\}$ and $K' = \{b\}$, then it is clear that $K \cup K' = \{a, b\}$ does not satisfy $\mathcal{E}$ while both K and K' do.

Note some essential differences between forbidding and enforcing: a forbidder describes a (finite) group of *sub*words that should not occur together in a language consistent with this forbidder, whereas an enforcer gives a relation between two (finite) groups of *words* in a language satisfying this enforcer. As a consequence of this, a single forbidder may cause the *absence* of *infinitely* many words, while a single enforcer may cause the *presence* of one of only *finitely* many words.

10.2.2 Evolving through enforcing

Let K_0 be a language, $\mathcal{E}$ an enforcing set, and assume that it is not true that $K_0\, \underline{\text{sat}}\, \mathcal{E}$, i.e., there are $(X, Y) \in \mathcal{E}$ with $X \subseteq K_0$ while $Y \cap K_0 = \varnothing$. Now add, for each of these K_0-violators (X, Y), at least one element of Y to K_0, and denote by K_1 the (possibly infinite, even when K_0 is finite) superset of K_0 constructed non-deterministically in this way. Then clearly none of the K_0-violators is a K_1-violator, but some enforcers that were not applicable to K_0 may be applicable to K_1 and thus may become K_1-violators. If so, then repeat the construction described above, and so on. This iterative 'repair procedure' is illustrated in Figure 10.1.

The underlying idea of this 'evolving procedure' is formalized as follows, in a more general way, that reflects the fact that when such an 'enforcing reaction' takes place in reality (i.e., in nature or in a laboratory), the premises of the reaction may be consumed (i.e., disappear) during the creation of the consequences.

Definition 10.3 For an enforcing set $\mathcal{E}$ and languages K and K' we say that K' is an $\mathcal{E}$-*extension* of K, written $K \vdash_{\mathcal{E}} K'$, if $X \subseteq K$ implies $K' \cap Y \neq \varnothing$, for each $(X, Y) \in \mathcal{E}$. □

Hence in general it is not necessarily the case that $K \subseteq K'$, as was the case for the procedure described above.

The $\mathcal{E}$-extension relation expresses the basic computation step induced by $\mathcal{E}$: a molecular system that has to satisfy $\mathcal{E}$ evolves according to $\vdash_{\mathcal{E}}$. It is also our basic notion for studying computations in the forbidding-enforcing systems that we describe later.

K_0

K_0 Z_0 K_1

K_0-violators are satisfied by $K_1 = K_0 \cup Z_0$

K_1 Z_1 K_2

K_1-violators are satisfied by $K_2 = K_1 \cup Z_1$

$\vdots \quad \vdots \quad \vdots$

Figure 10.1: Iterative repair procedure

The following proposition from [ER, EH$^+$00] says that the iterative repair procedure above yields the desired result.

Proposition 10.2 *Let $\mathcal{E}$ be an enforcing set and let $K_1, K_2, \dots$ be an infinite ascending sequence of languages. If $K_i \vdash_{\mathcal{E}} K_{i+1}$ for each $i \geq 1$, then $(\bigcup_{i\geq 1} K_i)$ $\underline{sat}$ $\mathcal{E}$.*

It is instructive to see that the previous result in general does not hold for finite ascending sequences: take for instance $\mathcal{E} = \{(\{a\},\{b\}),(\{a,b\},\{c\})\}$, and let $K_1 = \{a\}$ while $K_2 = \{a,b\}$. Then $K_1 \subseteq K_2$ and $K_1 \vdash_{\mathcal{E}} K_2$ whereas $K_1 \cup K_2$ does not satisfy $\mathcal{E}$. This agrees with our intuition: the molecular reactions go on all the time, providing that the needed components (molecules) are available. Thus such reactions may lead out of a finite language.

10.2.3 A finitary normal form

In this subsection we consider a normal form for enforcing sets. We begin by distinguishing two finiteness properties of enforcing sets.

For a finite language Z we define $\mathcal{E}(Z) = \{(X,Y) \in \mathcal{E} \mid X = Z\}$.

Definition 10.4

(1) An enforcing set $\mathcal{E}$ is *finitary* if, for each finite language Z, $\mathcal{E}(Z)$ is finite.

(2) An enforcing set $\mathcal{E}$ is *weakly finitary* if for each finite language K_1 there exists a finite language K_2 such that $K_1 \vdash_{\mathcal{E}} K_2$. $\square$

If an enforcing set $\mathcal{E}$ is finitary, it means that, for each premise set, $\mathcal{E}$ contains only a finite number of different enforcers having this premise set. This is a *syntactic* feature of $\mathcal{E}$. On the other hand, the property of being weakly finitary is more of a *semantic* property – it says something about the effect that the

enforcing specified by $\mathcal{E}$ has on finite languages. This effect is required to be 'continuous': each finite language *can* always evolve according to $\mathcal{E}$ into a finite language. Thus one can start with a finite language and evolve it according to $\mathcal{E}$ in a smooth way without 'exploding in one step' into an infinite set.

The basic relationship between finitary and weakly finitary enforcing sets is given by the following result ([ER, EH$^+$00]).

Proposition 10.3
(1) Every finitary enforcing set is weakly finitary.
(2) There exist weakly finitary enforcing sets that are not finitary.

The fact that every finitary enforcing set is also weakly finitary follows from the definitions.

The non-finitary enforcing set $\mathcal{E} = \{(\varnothing, \{a, b^n\}) \mid n \geq 1\}$ illustrates fact (2) above: clearly, for any finite language K, we have $K \vdash_{\mathcal{E}} K \cup \{a\}$, hence $\mathcal{E}$ is weakly finitary.

Another result from [ER, EH$^+$00] says that languages K that satisfy finitary enforcing sets $\mathcal{E}$ play for their finite subsets the role of the universe (Σ^*), meaning that each finite subset of K can evolve according to $\mathcal{E}$ to another finite subset of K.

Proposition 10.4 *Let $\mathcal{E}$ be a finitary enforcing set, and let K be a language such that K $\underline{\text{sat}}$ $\mathcal{E}$. For every finite language $L \subseteq K$, there exists a finite language $L' \subseteq K$ such that $L \vdash_{\mathcal{E}} L'$.*

Note that the above result does not hold if we require that $\mathcal{E}$ is weakly finitary rather than finitary. To see this consider again the weakly finitary enforcing set $\mathcal{E} = \{(\varnothing, \{a, b^n\}) \mid n \geq 1\}$. Let $K = \{b^n \mid n \geq 1\}$, then obviously K $\underline{\text{sat}}$ $\mathcal{E}$. Now let $L \subseteq K$ be finite and assume that $L \vdash_{\mathcal{E}} L'$ for a finite language $L' \subseteq K$. Let $m = \max\{\, n \mid b^n \in L'\}$ and consider the enforcer $E = (\varnothing, \{a, b^{m+1}\})$. Obviously $L' \cap \{a, b^{m+1}\} = \varnothing$, contradicting $L \vdash_{\mathcal{E}} L'$. Hence L' cannot be finite if it has to be a subset of K.

The following theorem is one of the main results of the forbidding-enforcing theory ([ER, EH$^+$00]).

Proposition 10.5 *For every enforcing set there exists an equivalent finitary enforcing set.*

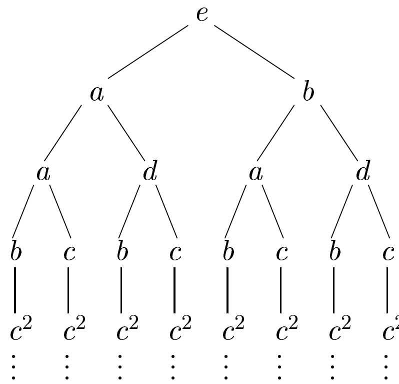
We show the idea behind the construction described in [ER] to create an equivalent finitary enforcing set for a given $\mathcal{E}$. That construction is rather sophisticated, because superfluous enforcers that may result from it are prevented beforehand. We use here a brute force technique and remove superfluous enforcers afterwards. We see this brute force technique as the idea behind the construction in [ER].

Example 10.7 Let $\Sigma = \{a, b, c, d, e\}$, and let $\mathcal{E}$ be the enforcing set

$$\{\,(\{e\},\{a,b\})\,,\,(\{e\},\{a,d\})\,,\,(\{e\},\{b,c\})\,\}\cup\{\,(\{e\},\{c^n\})\mid n\geq 2\}.$$

Note that $\mathcal{E}$ is satisfied by any language not containing e.

We create a finitary equivalent of $\mathcal{E}$ by trying to satisfy the enforcers from $\mathcal{E}$ one by one, in an arbitrary but fixed order (we choose the order as written above). Because we use such an order, when we have to satisfy the ith enforcer in it, for an $i \geq 2$, we can use the history of choices that we made to satisfy the $i-1$ previous enforcers. This history can be represented in a tree.



The construction is as follows. A language K that satisfies $\mathcal{E}$ should at least satisfy $(\{e\},\{a,b\})$. Assuming that this is the case – i.e., assuming that if K contains the word e, then it also contains a or b – the next enforcer in $\mathcal{E}$, $(\{e\},\{a,d\})$, can be replaced by two new enforcers: $(\{e,a\},\{a,d\})$ and $(\{e,b\},\{a,d\})$.

Then we assume that $(\{e,a\},\{a,d\})$ and $(\{e,b\},\{a,d\})$ are also satisfied. In other words, we assume that K contains the words e and a (and a), or the words e, a and d, or the words e, b and a, or the words e, b and d. Now we can replace the enforcer $(\{e\},\{b,c\}) \in \mathcal{E}$ by four new ones: $(\{e,a\},\{b,c\})$, $(\{e,a,d\},\{b,c\})$, $(\{e,b,a\},\{b,c\})$ and $(\{e,b,d\},\{b,c\})$.

In the next step we have to satisfy $(\{e\},\{c^2\})$ while assuming that the seven enforcers constructed above are satisfied. Therefore we add the eight enforcers $(\{e,a,b\},\{c^2\})$, $(\{e,a,c\},\{c^2\})$, $(\{e,a,d,b\},\{c^2\})$, $(\{e,a,d,c\},\{c^2\})$, $(\{e,b,a\},\{c^2\})$, $(\{e,b,a,c\},\{c^2\})$, $(\{e,b,d\},\{c^2\})$ and $(\{e,b,d,c\},\{c^2\})$ (actually these are seven enforcers, since $(\{e,a,b\},\{c^2\})$ is constructed for two reasons). And so on for $(\{e\},\{c^3\})$, etcetera.

Of the new enforcers introduced above, several can be removed, either because they are trivial (the enforcer $(\{e,a\},\{a,d\})$ is trivial, for instance, since if $\{e,a\} \subseteq L$ for some language L, then automatically $\{a,d\} \cap L \neq \varnothing$), or because their goal is already achieved by another enforcer (e.g., $(\{e,a,d\},\{b,c\})$

is superfluous because of $(\{e,a\},\{b,c\})$). In Section 11.2.2 we describe when enforcers may be removed in general.

Now let $\mathcal{E}'$ consist of the enforcers constructed above that are neither trivial nor superfluous, i.e.,

$$(\{e\},\{a,b\})\,,\ (\{e,b\},\{a,d\})\,,\ (\{e,a\},\{b,c\}),$$
$$(\{e,a,b\},\{c^2\})\,,\ (\{e,a,c\},\{c^2\})\,,\ (\{e,b,d\},\{c^2\})\,,\ \ldots$$

Then $\mathcal{E}'$ is equivalent to $\mathcal{E}$ and finitary (but still infinite; in fact, there is no equivalent finite enforcing set, by the proof of Lemma 11.12). □

10.3 Combining forbidding and enforcing

10.3.1 Definitions and examples

Definition 10.5 A *forbidding-enforcing system* (fe system for short) is a construct $\Gamma = (\mathcal{F}, \mathcal{E})$, where $\mathcal{F}$ is a forbidding set and $\mathcal{E}$ is an enforcing set. □

The corresponding *forbidding-enforcing family* (fe family), denoted $\mathcal{L}(\mathcal{F},\mathcal{E})$, consists of all languages that are both $\mathcal{F}$-consistent and $\mathcal{E}$-satisfying. Hence $\mathcal{L}(\mathcal{F},\mathcal{E}) = \mathcal{L}(\mathcal{F}) \cap \mathcal{L}(\mathcal{E})$.

Example 10.8 Let $\Sigma = \{a,b\}$ and let $\Gamma = (\mathcal{F},\mathcal{E})$ be the fe system obtained by combining the forbidding set $\mathcal{F} = \{\{aa,bb\},\{ab,ba\}\}$ from Example 10.1 and the enforcing set $\mathcal{E} = \{(\varnothing,\{b^n\}) \mid n \text{ is even}\}$ from Example 10.6. Then a language $K \subseteq \Sigma^*$ is in $\mathcal{L}(\mathcal{F},\mathcal{E})$ if and only if $K = K' \cup \{b^n \mid n \text{ is even}\}$ where either $K' \subseteq ab^* \cup b^*$ or $K' \subseteq b^*a \cup b^*$. □

Similar to the situation with forbidding and enforcing sets, we have the following property: if $\mathcal{F}' \subseteq \mathcal{F}$ and $\mathcal{E}' \subseteq \mathcal{E}$, then $\mathcal{L}(\mathcal{F},\mathcal{E}) \subseteq \mathcal{L}(\mathcal{F}',\mathcal{E}')$. Another fact that can easily be verified is the equality $\mathcal{L}(\mathcal{F}\cup\mathcal{F}',\mathcal{E}\cup\mathcal{E}') = \mathcal{L}(\mathcal{F},\mathcal{E}) \cap \mathcal{L}(\mathcal{F}',\mathcal{E}')$ for any two forbidding sets $\mathcal{F},\mathcal{F}'$ and any two enforcing sets $\mathcal{E},\mathcal{E}'$.

Using the closure properties of $\mathcal{L}(\mathcal{E})$ mentioned before it is clear that $\mathcal{L}(\mathcal{F},\mathcal{E})$ is not closed under union or intersection.

We carry over the 'finitary' qualification of enforcing sets to fe systems in the obvious way: an fe system $\Gamma = (\mathcal{F},\mathcal{E})$ is *finitary* if $\mathcal{E}$ is finitary.

We now give some extensive examples to illustrate the general concept of forbidding and enforcing. The examples discuss the representation of DNA molecules, the formalization of the splicing operation, and the satisfiability and Hamiltonian path problem.

Example 10.9 DNA molecules, single or (partially) double stranded, can be coded over a suitable alphabet of base pairs. For the matching base pairs we may use the symbols $\binom{\mathsf{a}}{\mathsf{t}}, \binom{\mathsf{t}}{\mathsf{a}}, \binom{\mathsf{c}}{\mathsf{g}}, \binom{\mathsf{g}}{\mathsf{c}}$. For the single stranded pieces we can use $\binom{\mathsf{a}}{.}, \binom{\mathsf{t}}{.}, \binom{\mathsf{c}}{.}, \binom{\mathsf{g}}{.}$ (upper strand) and $\binom{.}{\mathsf{a}}, \binom{.}{\mathsf{t}}, \binom{.}{\mathsf{c}}, \binom{.}{\mathsf{g}}$ (lower strand). For this example we consider languages over the alphabet Σ_{base} consisting of these twelve symbols.

We will give some natural requirements on a formal language representing the set of linear DNA molecules. These requirements can be formulated in our forbidding-enforcing framework.

First, of course, no molecule can have an unmatched base in the upper strand next to an unmatched base in the lower strand. This leads to forbidders $\{ \binom{\sigma}{.}\binom{.}{\tau} \}$ and $\{ \binom{.}{\sigma}\binom{\tau}{.} \}$ for each $\sigma, \tau \in \{\mathsf{a}, \mathsf{t}, \mathsf{c}, \mathsf{g}\}$.

Moreover, note that the molecule denoted by the string $\binom{\mathsf{a}}{\mathsf{t}}\binom{\mathsf{c}}{\mathsf{g}}\binom{\mathsf{c}}{.}\binom{\mathsf{c}}{\mathsf{g}}\binom{\mathsf{t}}{.}$ is also denoted by the inverted string $\binom{.}{\mathsf{t}}\binom{\mathsf{g}}{\mathsf{c}}\binom{.}{\mathsf{c}}\binom{\mathsf{g}}{\mathsf{c}}\binom{\mathsf{t}}{\mathsf{a}}$. We use $\underline{\text{inv}}$ to denote the operation of inversion, which is the composition of mirror image and replacing each symbol $\binom{\tau}{\sigma}$ by $\binom{\sigma}{\tau}$. Thus, in general, if x denotes a molecule α, then also $\underline{\text{inv}}\,(x)$ denotes α. Consequently, we need an infinite number of enforcers: $(\{x\}, \{\underline{\text{inv}}\,(x)\})$ for all $x \in \Sigma_{base}^+$.

The above set of forbidders and the set of enforcers together yield an fe system that admits only *correct* and *all* correct descriptions of linear DNA molecules. Hence we have: $K \subseteq \Sigma_{base}^+$ is in the fe family defined by the above forbidders and enforcers if and only if K is a correct description of a set of linear DNA molecules.

The effect of cutting such molecules by restriction enzymes (see Chapter 1) can easily be translated into enforcing rules. E.g., for the restriction enzyme *TaqI* (see, e.g., [NEB]) we have the enforcer $(\{x\binom{\mathsf{t}}{\mathsf{a}}\binom{\mathsf{c}}{\mathsf{g}}\binom{\mathsf{g}}{\mathsf{c}}\binom{\mathsf{a}}{\mathsf{t}}y\}, \{x\binom{\mathsf{t}}{\mathsf{a}}\binom{.}{\mathsf{g}}\binom{.}{\mathsf{c}}\})$ for each pair $x, y \in \Sigma_{base}^+$. Note that we have enforced only one of the two halves, the other follows by the palindromicity of *TaqI* and the inversion enforced above.

Recombination is then modelled by reversing the rules. E.g., ligating two pieces with overhang gc (one resulting from cutting with *TaqI* and the other a sticky end produced by the restriction enzyme *NarI* (see, e.g., [NEB])) can be enforced by

$$\left(\{x\binom{\mathsf{t}}{\mathsf{a}}\binom{.}{\mathsf{g}}\binom{.}{\mathsf{c}}, \binom{\mathsf{c}}{.}\binom{\mathsf{g}}{.}\binom{\mathsf{c}}{\mathsf{g}}\binom{\mathsf{c}}{\mathsf{g}}y\}, \{x\binom{\mathsf{t}}{\mathsf{a}}\binom{\mathsf{c}}{\mathsf{g}}\binom{\mathsf{g}}{\mathsf{c}}\binom{\mathsf{c}}{\mathsf{g}}\binom{\mathsf{c}}{\mathsf{g}}y\}\right).$$

□

Example 10.10 In splicing systems (Part I) the above operations are abstracted to the notion of splicing rules. The effect of a splicing rule (u_1, v_1, u_2, v_2), which says that two words $x_1u_1v_1y_1$ and $x_2u_2v_2y_2$ can be spliced to form the word $x_1u_1v_2y_2$, can be described by an enforcing set as follows:

$$\{(\{x_1u_1v_1y_1, x_2u_2v_2y_2\}, \{x_1u_1v_2y_2\}) \mid x_1, x_2, y_1, y_2 \in \Sigma^*\}.$$

More attractively, and more directly, one may have the rules of the form $(\{x, y, r\}, \{w\})$ where r is the restriction enzyme (after all it is a molecule) and x and y are spliced into w according to r. This seems to be attractive because, as various molecules are created during the evolution of such a system, new (restriction) enzymes may become available and so their effects will also be produced – in this way we can deal with dynamically changing sets of rules.

A splicing rule (u_1, v_1, u_2, v_2) may be specified in its usual string representation $u_1\#v_1\$u_2\#v_2$, whereas an enzyme may be given by its amino acid encoding, hence by a word over an alphabet of 20 symbols. □

Example 10.11 We explain how to describe an instance of the satisfiability problem by an fe system. Let $\Psi = (\neg x_1 \vee x_3 \vee x_6) \wedge (\neg x_1 \vee x_2 \vee \neg x_6)$ be a Boolean formula in 3-conjunctive normal form (see, e.g., [GJ79]). It consists of two clauses, each of which has to be satisfied.

The two possible truth assignments to the variable x_i can be encoded as the strings $\mathtt{x}\underline{\mathrm{bin}}\,(i)\mathtt{f}$ for 'x_i is false' and $\mathtt{x}\underline{\mathrm{bin}}\,(i)\mathtt{t}$ for 'x_i is true', where $\underline{\mathrm{bin}}\,(i) \in \{\mathtt{0},\mathtt{1}\}^*$ is a suitable binary encoding of i. Any language coding a truth assignment must have exactly one of the assignments for each variable. This can be achieved by having the brute enforcers $(\varnothing, \{\mathtt{x}v\mathtt{f}, \mathtt{x}v\mathtt{t}\})$ for each $v \in \{\mathtt{0},\mathtt{1}\}^*$ (this constitutes the enforcing set $\mathcal{E}_U$), and by having the forbidder $\{\mathtt{x}v\mathtt{f}, \mathtt{x}v\mathtt{t}\}$ for each $v \in \{\mathtt{0},\mathtt{1}\}^*$ (this constitutes the forbidding set $\mathcal{F}_U$).

Besides these universally valid restrictions, the formula Ψ itself places additional restrictions on the truth assignment. For instance, the clause $\neg x_1 \vee x_3 \vee x_6$ demands to assign true either to $\neg x_1$, or to x_3, or to x_6. Equivalently, it demands not to assign false to all of $\neg x_1$, x_3, and x_6 at the same time, i.e., it forbids the words $\mathtt{x1t}$, $\mathtt{x11f}$, and $\mathtt{x110f}$ to occur at the same time. Hence, the first clause is represented by the forbidder $\{\mathtt{x1t}, \mathtt{x11f}, \mathtt{x110f}\}$ and the second clause by the forbidder $\{\mathtt{x1t}, \mathtt{x10f}, \mathtt{x110t}\}$.

In this way, representing each clause by a forbidder, we get the forbidding set $\mathcal{F}_\Psi$. Now the 'universal' forbidders and enforcers together with the forbidders defined by the formula Ψ yield the fe system $\Gamma_\Psi = (\mathcal{F}_U \cup \mathcal{F}_\Psi, \mathcal{E}_U)$. This Γ_Ψ provides a succinct representation for the satisfiability of Ψ: Ψ is satisfiable if and only if $\mathcal{L}(\Gamma_\Psi)$ is non-empty. □

Example 10.12 We demonstrate how to transform an instance of the Hamiltonian Path Problem into an fe system, in such a way that each language in the corresponding fe family consists of exactly one solution to this instance.

Let $G = (V, E)$ be a directed graph with n nodes, and let x_{in} and x_f be designated initial and final nodes of G. A Hamiltonian path of G is a path from x_{in} to x_f that visits each node of G exactly once. Formally, it is a sequence $x_1, \ldots, x_n$ of nodes of G such that $x_1 = x_{in}$, $x_n = x_f$, $x_i \neq x_j$ for $i \neq j$, and $(x_i, x_{i+1}) \in E$ for $1 \leq i < n$.

Consider the alphabet $\Delta = \{\underline{1}, \ldots, \underline{n}\}$, and assume that $V = \Delta$. A path $x_1, \ldots, x_n$ of G, where $x_i \in V$ for all $1 \leq i \leq n$, is coded as the finite language $\{\underline{1}x_1, \underline{2}x_2, \ldots, \underline{n}x_n\} \subseteq \Delta^2$, meaning that x_i is the ith node on the path.

A forbidding-enforcing system $(\mathcal{F}, \mathcal{E})$ for which every language in the corresponding fe family is an encoding of a Hamiltonian path in G is constructed as follows.

First we ensure that each possible solution starts in x_{in} and ends in x_f, by putting $(\varnothing, \{\underline{1}x_{in}\})$ and $(\varnothing, \{\underline{n}x_f\})$ in $\mathcal{E}$. Then, we guarantee that each possible solution consists of a permutation of the nodes of G, by adding, for every $1 < i < n$, the enforcer $(\varnothing, \{\underline{i}x \mid x \in V - \{x_{in}, x_f\}\})$ to $\mathcal{E}$, and the forbidders $\{\underline{i}x, \underline{j}x\}$ for each $i \neq j$ where $1 \leq i, j \leq n$ to $\mathcal{F}$. Finally we forbid consecutive pairs of nodes between which there is no edge in G through the forbidders $\{\underline{i}x, \underline{i{+}1}y\}$ for $(x, y) \notin E$ and $1 \leq i < n$.

Hence there exists a Hamiltonian path in G if and only if $\mathcal{L}(\mathcal{F}, \mathcal{E}) \neq \varnothing$. □

10.3.2 The structure of computation in fe systems

We move now to consider the structure of computations in fe systems, and in particular we claim that for finitary fe systems there is an elegant representation, in the form of a tree, of all the computations in such a system.

We use here the standard notion of a tree. The trees are rooted, node-labelled, they may be infinite but are always finitely branching. This means that each node has only a finite number of children (however, we do not assume that there is a common bound on the number of children for each node). The label of a node v in a tree τ is denoted by $\underline{\text{lab}}_\tau(v)$. We call a path in a tree a *full* path if it starts at the root and either ends at a leaf or is infinite.

Definition 10.6 Let $\Gamma = (\mathcal{F}, \mathcal{E})$ be an fe system, and let τ be a tree.
Then τ is a Γ-*tree* if
(1) each node label is an element of $\mathcal{L}_{\text{fin}}(\mathcal{F})$,
(2) if a node v_2 is a descendant of a node v_1, then $\underline{\text{lab}}_\tau(v_1) \subseteq \underline{\text{lab}}_\tau(v_2)$ and $\underline{\text{lab}}_\tau(v_1) \vdash_{\mathcal{E}} \underline{\text{lab}}_\tau(v_2)$. □

Hence the influence of the forbidding set is expressed by the sort of node labels that are admitted, while the influence of the enforcing set is expressed through the condition on the sort of languages that can follow each other on a single path – this is illustrated in Figure 10.2.

We now consider a Γ-tree where all the languages from $\mathcal{L}(\Gamma)$, finite and infinite, are represented.

Definition 10.7 Let $\Gamma = (\mathcal{F}, \mathcal{E})$ be an fe system. A Γ-tree τ is *complete* if
(1) if $K \in \mathcal{L}(\Gamma)$ is finite, then K is a node label of τ,

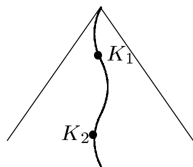


Figure 10.2: $K_1, K_2 \in \mathcal{L}_{\text{fin}}(\mathcal{F})$, $K_1 \subseteq K_2$, and $K_1 \vdash_{\mathcal{E}} K_2$

(2) if $K \in \mathcal{L}(\Gamma)$ is infinite, then there exists an infinite path π in τ such that $K = \bigcup_{v \in \pi} \underline{\text{lab}}_\tau(v)$. □

We know already that finitary fe systems constitute a normal form for forbidding-enforcing systems meaning that as far as the specifications of fe families are concerned one can restrict oneself to finitary fe systems. However, the real attraction of finitary fe systems stems from the following result ([ER, EH$^+$00]).

Proposition 10.6 *For each finitary fe system* Γ *there exists a complete* Γ*-tree.*

This means that every finitary fe system Γ can be 'completely' represented by a complete Γ-tree τ, that represents both *all languages* defined by Γ and *all computations* taking place within Γ:
(1) all finite languages in $\mathcal{L}(\Gamma)$ occur as node labels in τ,
(2) by taking for each infinite path the union of all languages along this path we get all infinite languages in $\mathcal{L}(\Gamma)$,
(3) by following all full paths in τ we get all evolving computations of Γ.

Since finitary fe systems form a normal form for fe systems, one can represent all languages defined by an arbitrary fe system by a tree, viz., the Γ-tree of an equivalent finitary fe system Γ. What will not carry over is the structure of computations in the original fe system; in particular, the label of a node in the Γ-tree is not necessarily an $\mathcal{E}$-extension of the label of its parent, where $\mathcal{E}$ is the enforcing set of the original fe system.

Proposition 10.7 *For each fe system* Γ *there exists a finitely branching tree* τ *with nodes labelled by finite languages such that for each language* K, $K \in \mathcal{L}(\Gamma)$ *iff there exists a full path* π *in* τ *such that* $K = \bigcup_{v \in \pi} \underline{lab}_\tau(v)$.

10.4 Research topics

Since forbidding-enforcing is a new model of computation, there are many standard (formal language theory) issues to investigate. In Chapter 11 we discuss, among other subjects, finite versus infinite forbidding and enforcing sets, normal forms, and deterministic versus non-deterministic enforcing sets.

A number of results concerning fe systems deal with the fact that for an infinite ascending sequence of languages satisfying certain properties their union satisfies the same, or closely related, properties. In Chapter 12 we generalise Propositions 10.1(3) and 10.2, that both deal with sequences of languages in forbidding or enforcing families, and we discuss the different role that finite languages play in the forbidding-enforcing model when compared to standard formal language theory (grammars and automata).

Chapter 11

Properties of forbidding sets and enforcing sets

We discuss some basic formal language properties of forbidding sets and enforcing sets: finiteness versus infinity, normal forms, and determinism versus non-determinism in enforcing sets.

11.1 Forbidding sets

11.1.1 Finite forbidding sets

Consider the forbidding set $\mathcal{F}_1 = \{\{ab\}, \{a^2b^2\}, \{a^3b^3\}, \ldots\}$. Clearly, if a language does not contain the subword ab, then it also does not contain the subwords a^ib^i for each $i \geq 2$, since they all contain ab as a subword. Consequently $\mathcal{F}_1$ is equivalent to the finite forbidding set $\mathcal{F}_2 = \{\{ab\}\}$.

Not every infinite forbidding set has a finite equivalent, as demonstrated by the following theorem.

Theorem 11.1 *There are forbidding sets for which there is no equivalent finite forbidding set.*

Proof. For a finite forbidding set $\mathcal{F}$, let $\ell = \max\{|w| \mid w \in \bigcup_{F \in \mathcal{F}} F\}$. Now, if for two words x and y it is the case that $\underline{\text{sub}}(x)|_{\leq \ell} = \underline{\text{sub}}(y)|_{\leq \ell}$, then $\mathcal{F}$ cannot distinguish between x and y. In other words, for each $F \in \mathcal{F}$ it holds that $F \not\subseteq \underline{\text{sub}}(x)$ if and only if $F \not\subseteq \underline{\text{sub}}(y)$, hence $\{x\} \in \mathcal{L}(\mathcal{F})$ if and only if $\{y\} \in \mathcal{L}(\mathcal{F})$.

Now consider $\mathcal{F}_{\text{even}} = \{\{ab^2a\}, \{ab^4a\}, \{ab^6a\}, \ldots\}$. For each ℓ, the words $x = ab^\ell a$ and $y = ab^{\ell+1}a$ differ only on subwords of length greater than ℓ. Clearly $\{x\} \in \mathcal{L}(\mathcal{F}_{\text{even}})$ if and only if $\{y\} \notin \mathcal{L}(\mathcal{F}_{\text{even}})$. Hence $\mathcal{L}(\mathcal{F}_{\text{even}}) \neq \mathcal{L}(\mathcal{F})$ for all finite $\mathcal{F}$. $\square$

11.1.2 Two useful normal forms

We started the previous subsection with an example of a forbidding set that could be reduced in size by removing certain parts of it. In this subsection we formalize the conditions under which we can do the same for arbitrary forbidding sets, i.e., we discuss how redundancy can be removed from an arbitrary forbidding set without changing the family of consistent languages.

One kind of redundancy occurs within forbidders, as described in the following example and lemma.

Example 11.1 Consider the forbidder $\{a, b, ab\}$. A language K is consistent with $\{a, b, ab\}$ if $a \notin \underline{\text{sub}}(K)$ or $b \notin \underline{\text{sub}}(K)$ or $ab \notin \underline{\text{sub}}(K)$. Clearly in each of these three cases ab cannot be a subword of K, hence $K \underline{\text{con}} \{a, b, ab\}$ implies $K \underline{\text{con}} \{ab\}$. Conversely, if $ab \notin \underline{\text{sub}}(K)$, then obviously $\{a, b, ab\} \not\subseteq \underline{\text{sub}}(K)$. Consequently $K \underline{\text{con}} \{a, b, ab\}$ if and only if $K \underline{\text{con}} \{ab\}$, and we may replace $\{a, b, ab\}$ by its subset $\{ab\}$. □

The crucial point in the above example is the fact that a and b are subwords of ab. In other words, the forbidder $\{a, b, ab\}$ is not subword free and therefore contains redundancy. This redundancy can be removed very easily, as shown by the following lemma.

We call a forbidding set *subword free* if all its forbidders are subword free.

Lemma 11.2 *For every forbidding set there exists an equivalent subword free forbidding set.*

Proof. Given a forbidding set $\mathcal{F}$, consider $\mathcal{F}' = \{\underline{\text{sub}}_{\text{max}}(F) \mid F \in \mathcal{F}\}$, then it is obvious that every $F' \in \mathcal{F}'$ is subword free. Moreover, it is clear that, for all $F \in \mathcal{F}$ and all languages K, $F \subseteq \underline{\text{sub}}(K)$ if and only if $\underline{\text{sub}}_{\text{max}}(F) \subseteq \underline{\text{sub}}(K)$. Consequently $\mathcal{L}(\mathcal{F}) = \mathcal{L}(\mathcal{F}')$. □

Another way of reducing redundancy is to remove superfluous forbidders, like in the example given in Subsection 11.1.1. The crucial point there is the fact that the forbidder $\{ab\}$ consists of a subword of each of the other forbidders. This observation can be generalised as follows.

Lemma 11.3 *Let $\mathcal{F}$ be a forbidding set, and let $F_1, F_2 \in \mathcal{F}$ with $F_1 \neq F_2$. If $\underline{\text{sub}}(F_1) \subseteq \underline{\text{sub}}(F_2)$, then $\mathcal{F} \sim \mathcal{F} - \{F_2\}$.*

Proof. It is clear that $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F} - \{F_2\})$. Moreover, $F_1 \not\subseteq \underline{\text{sub}}(K)$ for a language K implies $\underline{\text{sub}}(F_1) \not\subseteq \underline{\text{sub}}(K)$. Since $\underline{\text{sub}}(F_1) \subseteq \underline{\text{sub}}(F_2)$ then also $\underline{\text{sub}}(F_2) \not\subseteq \underline{\text{sub}}(K)$ which implies $F_2 \not\subseteq \underline{\text{sub}}(K)$, and thus $\mathcal{L}(\mathcal{F} - \{F_2\}) \subseteq \mathcal{L}(\mathcal{F})$. Consequently $\mathcal{F} \sim \mathcal{F} - \{F_2\}$. □

As an aside, note that it may be that $\underline{\text{sub}}(F) = \underline{\text{sub}}(F')$, for two different forbidders F and F' (this is the case for, for instance, the forbidders $\{a, ab\}$ and $\{b, ab\}$). Also note that this cannot occur when both F and F' are subword free, since then $\underline{\text{sub}}(F) = \underline{\text{sub}}(F')$ if and only if $F = F'$ (Lemma 2.1).

We can generalise Lemma 11.3 to removing a (possibly infinite) subset of a forbidding set, instead of just one element.

Lemma 11.4 *Let $\mathcal{F}$ and $\mathcal{F}'$ be forbidding sets with $\mathcal{F}' \subseteq \mathcal{F}$ such that for each $F \in \mathcal{F}$ there is an $F' \in \mathcal{F}'$ with $\underline{sub}(F') \subseteq \underline{sub}(F)$. Then $\mathcal{F}' \sim \mathcal{F}$.*

Proof. Clearly $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}')$.

To prove that $\mathcal{L}(\mathcal{F}') \subseteq \mathcal{L}(\mathcal{F})$, note that the condition in the lemma means that for each $F \in \mathcal{F}$ there is an $F' \in \mathcal{F}'$ such that $K \,\underline{\text{con}}\, F'$ implies $K \,\underline{\text{con}}\, F$, for a language K. Consequently, if $K \,\underline{\text{con}}\, F'$ for all $F' \in \mathcal{F}'$, then also $K \,\underline{\text{con}}\, F$ for all $F \in \mathcal{F}$. $\square$

We define two different forbidders F_1 and F_2 to be *subword incomparable* if neither $\underline{\text{sub}}(F_1) \subseteq \underline{\text{sub}}(F_2)$ nor $\underline{\text{sub}}(F_2) \subseteq \underline{\text{sub}}(F_1)$. We call a forbidding set $\mathcal{F}$ subword incomparable if each pair of distinct forbidders F_1 and F_2 is subword incomparable. Note that if we have $\underline{\text{sub}}(F_1) \not\subseteq \underline{\text{sub}}(F_2)$ for all $F_1, F_2 \in \mathcal{F}$ with $F_1 \neq F_2$, then we also have $F_1 \not\subseteq F_2$ for all $F_1, F_2 \in \mathcal{F}$ with $F_1 \neq F_2$.

Lemma 11.5 *For every forbidding set there is an equivalent subword incomparable forbidding set.*

Proof. Let $\mathcal{F}$ be a forbidding set. Because of Lemma 11.2 we may assume that $\mathcal{F}$ is subword free. Now define $\mathcal{F}' = \{F' \in \mathcal{F} \mid$ there is no $F'' \in \mathcal{F}$ such that $F'' \neq F'$ and $\underline{\text{sub}}(F'') \subseteq \underline{\text{sub}}(F')\}$. Note that we need the subword freeness of $\mathcal{F}$ to ensure that also forbidders that are different but have the same subwords are represented in $\mathcal{F}'$ (by the set of maximal subwords of one of them).

It is clear from the definition that $\mathcal{F}'$ cannot contain two different forbidders F_1 and F_2 with $\underline{\text{sub}}(F_1) \subseteq \underline{\text{sub}}(F_2)$ or vice versa, hence $\mathcal{F}'$ is subword incomparable.

To prove that $\mathcal{F}' \sim \mathcal{F}$ we will apply Lemma 11.4. Observe that indeed $\mathcal{F}' \subseteq \mathcal{F}$, and that for each $F \in \mathcal{F}$ there is an $F' \in \mathcal{F}'$ with $\underline{\text{sub}}(F') \subseteq \underline{\text{sub}}(F)$: consider $F \in \mathcal{F}$. Either $F \in \mathcal{F}'$, or there exists an $F_1 \in \mathcal{F}$ such that $F_1 \neq F$ and $\underline{\text{sub}}(F_1) \subseteq \underline{\text{sub}}(F)$. Again, either $F_1 \in \mathcal{F}'$, or we can find an $F_2 \in \mathcal{F}$ with $F_2 \neq F_1$ and $\underline{\text{sub}}(F_2) \subseteq \underline{\text{sub}}(F_1)$. In this way we obtain a sequence $F_0, F_1, F_2 \dots$ in $\mathcal{F}$ such that $F_0 = F$, $F_{i+1} \neq F_i$ and $\underline{\text{sub}}(F_{i+1}) \subseteq \underline{\text{sub}}(F_i)$, for each $i \geq 0$. Because $\mathcal{F}$ is subword free, $F_{i+1} \neq F_i$ implies $\underline{\text{sub}}(F_{i+1}) \neq \underline{\text{sub}}(F_i)$, and consequently $\underline{\text{sub}}(F_{i+1}) \subset \underline{\text{sub}}(F_i)$. Since each $\underline{\text{sub}}(F_i)$ is finite, this implies that our construction ends in a finite number of steps, and we finally find a forbidder $F_k \in \mathcal{F}$ such that $F_k \in \mathcal{F}'$ and $\underline{\text{sub}}(F_k) \subseteq \underline{\text{sub}}(F)$. Hence Lemma 11.4 is applicable and thus we obtain $\mathcal{F}' \sim \mathcal{F}$. $\square$

If a forbidding set is subword incomparable, then no forbidder can be removed without changing the family of consistent languages.

Lemma 11.6 *Let $\mathcal{F}$ be a forbidding set.*
If $\mathcal{F}$ is subword incomparable, then $\mathcal{L}(\mathcal{F}) \subset \mathcal{L}(\mathcal{F} - \{F\})$ for every $F \in \mathcal{F}$.

Proof. It is obvious that $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F} - \{F\})$.

We prove the strictness of the inclusion by demonstrating that $F \in \mathcal{L}(\mathcal{F} - \{F\})$ but not $F \in \mathcal{L}(\mathcal{F})$, for an arbitrary forbidder $F \in \mathcal{F}$. Obviously, F is not consistent with itself, thus $F \notin \mathcal{L}(\mathcal{F})$.

Furthermore, since $\mathcal{F}$ is subword incomparable, $\underline{\text{sub}}\,(G) \not\subseteq \underline{\text{sub}}\,(F)$ for all $G \in \mathcal{F} - \{F\}$, and thus also $G \not\subseteq \underline{\text{sub}}\,(F)$ for all $G \in \mathcal{F} - \{F\}$. Consequently $F\,\underline{\text{con}}\,G$ for all $G \in \mathcal{F} - \{F\}$, and thus $F \in \mathcal{L}(\mathcal{F} - \{F\})$. □

11.1.3 Minimal forbidding sets

We say that a forbidding set $\mathcal{F}$ is in *minimal normal form* if $\mathcal{F}$ is both subword free and subword incomparable. Since obviously the forbidding set $\mathcal{F}'$ constructed in the proof of Lemma 11.5 has both these properties, 'minimal normal form' is indeed a normal form.

Lemma 11.7 *For each forbidding set there exists an equivalent forbidding set in minimal normal form.*

A forbidding set in minimal normal form is indeed minimal, or 'redundancy free', in the sense that removing one of its forbidders or even one element from one forbidder yields a forbidding set that is not equivalent to the original one. This follows from Lemma 11.6 and the following result.

Lemma 11.8 *Let $\mathcal{F}$ be a forbidding set that contains a forbidder $F = \{f_1, \ldots, f_n, w\}$ for some $n \geq 1$, with $w \neq f_i$ for all $1 \leq i \leq n$, and let $\mathcal{F}' = (\mathcal{F} - \{F\}) \cup \{F'\}$, where $F' = \{f_1, \ldots, f_n\}$. If $\mathcal{F}$ is in minimal normal form, then $\mathcal{L}(\mathcal{F}') \subset \mathcal{L}(\mathcal{F})$.*

Proof. To prove $\mathcal{L}(\mathcal{F}') \subseteq \mathcal{L}(\mathcal{F})$ we need that, for all languages K, $K \in \mathcal{L}(\mathcal{F}')$ implies $K \in \mathcal{L}(\mathcal{F})$. Since the only difference between $\mathcal{F}$ and $\mathcal{F}'$ is that $\mathcal{F}'$ contains F' instead of F, it suffices to prove that $K\,\underline{\text{con}}\,F'$ implies $K\,\underline{\text{con}}\,F$, which is true because from $F' \subseteq F$ it follows that $F' \not\subseteq \underline{\text{sub}}\,(K)$ implies $F \not\subseteq \underline{\text{sub}}\,(K)$.

It is clear that $F'\,\underline{\text{con}}\,\mathcal{F}'$ does not hold, because $F' \in \mathcal{F}'$, thus to prove that $\mathcal{L}(\mathcal{F}) - \mathcal{L}(\mathcal{F}') \neq \varnothing$, it suffices to demonstrate that $F'\,\underline{\text{con}}\,\mathcal{F}$, i.e., $F'\,\underline{\text{con}}\,F$ and $F'\,\underline{\text{con}}\,G$ for all $G \in \mathcal{F}$ with $G \neq F$.

For F' to be consistent with F, we need $F \not\subseteq \underline{\text{sub}}(F')$, i.e., $\{f_1, \ldots, f_n, w\} \not\subseteq \underline{\text{sub}}(\{f_1, \ldots, f_n\})$. Since F is subword free, we have $w \notin \underline{\text{sub}}(f_i)$ for all $1 \leq i \leq n$ and hence $w \notin \underline{\text{sub}}(\{f_1, \ldots, f_n\})$. Consequently $F' \, \underline{\text{con}} \, F$.

Now let $G \in \mathcal{F}$ with $G \neq F$. Then $F' \, \underline{\text{con}} \, G$ if and only if $G \not\subseteq \underline{\text{sub}}(F')$, which is the same as $\underline{\text{sub}}(G) \not\subseteq \underline{\text{sub}}(F')$. We know that $\underline{\text{sub}}(G) \not\subseteq \underline{\text{sub}}(F)$, because $\mathcal{F}$ is subword incomparable. Since $F' \subseteq F$ and thus $\underline{\text{sub}}(F') \subseteq \underline{\text{sub}}(F)$, we now have $\underline{\text{sub}}(G) \not\subseteq \underline{\text{sub}}(F')$. Consequently $F' \, \underline{\text{con}} \, G$ for all $G \in \mathcal{F}$ with $G \neq F$. Therefore $\mathcal{L}(\mathcal{F}') \subset \mathcal{L}(\mathcal{F})$. □

Moreover, for each forbidding set there is only one equivalent forbidding set that is in minimal normal form.

Lemma 11.9 *If two forbidding sets $\mathcal{F}$ and $\mathcal{F}'$ are both in minimal normal form and $\mathcal{F} \sim \mathcal{F}'$, then $\mathcal{F} = \mathcal{F}'$.*

Proof. It suffices to prove $\mathcal{F} \subseteq \mathcal{F}'$. Let $F \in \mathcal{F}$. Then $F \notin \mathcal{L}(\mathcal{F}) = \mathcal{L}(\mathcal{F}')$. Hence there is an $F' \in \mathcal{F}'$ such that $F' \subseteq \underline{\text{sub}}(F)$. Similarly, since $F' \in \mathcal{F}'$, there is an $F'' \in \mathcal{F}$ such that $F'' \subseteq \underline{\text{sub}}(F')$. Consequently $\underline{\text{sub}}(F'') \subseteq \underline{\text{sub}}(F)$, and, since $\mathcal{F}$ is subword incomparable, $F'' = F$. Hence $\underline{\text{sub}}(F) = \underline{\text{sub}}(F')$ and so (by Lemma 2.1, since $\mathcal{F}$ and $\mathcal{F}'$ are subword free) we have $F = F'$. Thus $F \in \mathcal{F}'$. □

Hence the conclusion of this subsection is the following.

Theorem 11.10 *For every forbidding set there exists a unique equivalent forbidding set in minimal normal form.*

Because of this theorem, and since a forbidding set in minimal normal form is indeed minimal, as demonstrated in Lemma's 11.6 and 11.8, we introduce a notation for it: $\underline{\text{min}}(\mathcal{F})$.

11.1.4 Maximal forbidding sets

For a forbidding set $\mathcal{F}$, we define the set

$$\underline{\text{max}}(\mathcal{F}) = \{K \mid K \text{ is finite and not } K \, \underline{\text{con}} \, \mathcal{F}\}.$$

Note that $\underline{\text{max}}(\mathcal{F})$ is also a forbidding set (i.e., a family of finite languages), and that, since each $F \in \mathcal{F}$ is not consistent with $\mathcal{F}$, it holds that $\mathcal{F} \subseteq \underline{\text{max}}(\mathcal{F})$. It is easy to argue that $\underline{\text{max}}(\mathcal{F})$ is maximal in the sense that it cannot be extended by adding a forbidder: in that case a new finite language is added to $\underline{\text{max}}(\mathcal{F})$. This language can be either consistent or inconsistent with $\mathcal{F}$. In the former case it violates the definition of $\underline{\text{max}}(\mathcal{F})$, whereas in the latter case it was already in $\underline{\text{max}}(\mathcal{F})$, since $\underline{\text{max}}(\mathcal{F})$ contains all finite languages that are not consistent with $\mathcal{F}$.

The following result shows that, for every $\mathcal{F}$, $\underline{\text{max}}(\mathcal{F})$ is equivalent to $\mathcal{F}$.

Lemma 11.11 *Let $\mathcal{F}$ be a forbidding set. Then $\mathcal{F} \sim \underline{\max}(\mathcal{F})$.*

Proof. Since $\mathcal{F} \subseteq \underline{\max}(\mathcal{F})$ it is clear that $\mathcal{L}(\underline{\max}(\mathcal{F})) \subseteq \mathcal{L}(\mathcal{F})$.

Now let $K \in \mathcal{L}(\mathcal{F})$, i.e., for all $F \in \mathcal{F}$ it holds that $F \not\subseteq \underline{\text{sub}}(K)$. We have to prove that $G \not\subseteq \underline{\text{sub}}(K)$ for all $G \in \underline{\max}(\mathcal{F})$. For each such G there is an $F' \in \mathcal{F}$ with $F' \subseteq \underline{\text{sub}}(G)$. Since $F' \not\subseteq \underline{\text{sub}}(K)$ it also holds that $\underline{\text{sub}}(G) \not\subseteq \underline{\text{sub}}(K)$, which is equivalent to $G \not\subseteq \underline{\text{sub}}(K)$. Hence $K \in \mathcal{L}(\underline{\max}(\mathcal{F}))$. □

The names of $\underline{\min}(\mathcal{F})$ and $\underline{\max}(\mathcal{F})$, for a forbidding set $\mathcal{F}$, are well chosen: if $\mathcal{F}$ is subword free (which is a normal form), then $\underline{\min}(\mathcal{F}) \subseteq \mathcal{F} \subseteq \underline{\max}(\mathcal{F})$, and as we have seen above, $\underline{\min}(\mathcal{F})$ and $\underline{\max}(\mathcal{F})$ are minimal and maximal, respectively, in the sense that nothing can be removed from $\underline{\min}(\mathcal{F})$ without changing the family of consistent languages, and no forbidder can be added to $\underline{\max}(\mathcal{F})$.

11.1.5 A tree representation for consistent families

Every language K can be described as a sequence of finite languages, namely $K = \bigcup_{i \geq 0} K|_{\leq i}$. Note that $K|_{\leq i} \subseteq K|_{\leq i+1}$ for every $i \geq 0$, and that, if K and L are different languages and ℓ is the length of a shortest word in $(K-L) \cup (L-K)$, then $K|_{\leq i} = L|_{\leq i}$ for all $0 \leq i < \ell$. Moreover, if $\mathcal{F}$ is a forbidding set, then it follows from Proposition 10.1(2) and (3) that $K\ \underline{\text{con}}\ \mathcal{F}$ if and only if $K|_{\leq i}\ \underline{\text{con}}\ \mathcal{F}$ for all $i \geq 0$ (see also Section 12.4).

All these observations together suggest a representation of $\mathcal{L}(\mathcal{F})$ by a tree (rooted, finitely branching, with possibly infinite paths), in which the nodes are labelled by finite languages consistent with $\mathcal{F}$, i.e., the node labels are elements of $\mathcal{L}_{\text{fin}}(\mathcal{F})$. The node labels on level ℓ are those languages in $\mathcal{L}_{\text{fin}}(\mathcal{F})$ that contain only words of length ℓ or smaller, for $\ell \geq 0$ (and the root of the tree has level 0). A node labelled X is the unique parent of a node with label Y if and only if $X \neq Y$ and $X = \{y \in Y \mid |y| \text{ is less than the length of the longest word in } Y\}$, i.e., if and only if X is extended to Y by adding words of one particular length that is greater than the length of the largest word in X.

11.2 Enforcing sets

11.2.1 Finite enforcing sets

In our approach we allow infinite enforcing sets. Unfortunately one cannot restrict oneself to finite enforcing sets only. This is caused by the fact that finite enforcing sets cannot have any effect on words longer than a certain length and thus have influence on only a finite number of words. Consequently the following lemma can be proved.

Lemma 11.12 *If $\mathcal{E}$ is a finite enforcing set, then $\mathcal{L}(\mathcal{E})$ contains a finite language.*

Proof. If $\mathcal{E}$ is a finite enforcing set, then we can define $n = \max\{|w| \mid w \in Y$ for some $(X,Y) \in \mathcal{E}\}$. Thus the finite language $\Sigma|_{\leq n}$ satisfies $\mathcal{E}$, because for every enforcer $(X,Y) \in \mathcal{E}$ it holds that $Y \cap \Sigma|_{\leq n} \neq \varnothing$. Consequently $\mathcal{L}(\mathcal{E})$ contains a finite language. □

Indeed, this lemma implies that finite enforcing sets are strictly less powerful than infinite enforcing sets.

Lemma 11.13 *There are enforcing sets for which there is no equivalent finite enforcing set.*

Proof. Let $a \in \Sigma$, and let $\mathcal{E} = \{(\varnothing, \{a\})\} \cup \{(\{w\}, \{wa\}) \mid w \in \Sigma^+\}$. Clearly, $a^+ \subseteq K$ for all $K \in \mathcal{L}(\mathcal{E})$. Consequently $\mathcal{L}(\mathcal{E})$ does not contain a finite language, thus by Lemma 11.12 cannot be defined by a finite enforcing set. □

11.2.2 Normal forms

Apart from the finitary normal form for enforcing sets described in Subsection 10.2.3, two other normal forms can be proved.

First, an enforcer may be superfluous on its own: enforcers (X,Y) with $X \cap Y \neq \varnothing$ are always satisfied and therefore can be omitted. These enforcers are called 'trivial' in Example 10.7.

Second, some enforcers make other enforcers superfluous, as shown below.

Lemma 11.14 *Let $\mathcal{E}$ be an enforcing set, and let (X,Y) and (X',Y') be two different enforcers in $\mathcal{E}$ with $X \subseteq X'$ and $Y \subseteq Y'$. Then $\mathcal{E} \sim \mathcal{E} - \{(X',Y')\}$.*

Proof. It is clear that $\mathcal{L}(\mathcal{E}) \subseteq \mathcal{L}(\mathcal{E} - \{(X',Y')\})$.

To prove that $\mathcal{L}(\mathcal{E} - \{(X',Y')\}) \subseteq \mathcal{L}(\mathcal{E})$ we show that $K \ \underline{\text{sat}}\ (X,Y)$ implies that $K \ \underline{\text{sat}}\ (X',Y')$, for (X,Y) and (X',Y') as in the statement of the lemma. If $X' \subseteq K$ then $X \subseteq K$, hence $Y \cap K \neq \varnothing$, and so $Y' \cap K \neq \varnothing$. □

Hence we may always assume that, for the enforcing set $\mathcal{E}$ under consideration, for each enforcer $(X,Y) \in \mathcal{E}$ it holds that $X \cap Y = \varnothing$, and for each pair (X,Y) and (X',Y') of enforcers it holds that $X \not\subseteq X'$ or $Y \not\subseteq Y'$.

11.2.3 Deterministic enforcing sets

In connection with Example 10.5 we mentioned the fact that the definition of enforcer allows non-determinism in the sense that, for an enforcer $E = (X,Y)$, if E is applicable to a language K but not satisfied by K, then to make K satisfy E it suffices to add any non-empty subset of Y to K. Obviously, there is no non-determinism involved if Y is a singleton.

Thus we call an enforcer (X,Y) *deterministic* if $|Y| = 1$, and we say that an enforcing set is deterministic if all its enforcers are deterministic.

Deterministic enforcing sets are less powerful than non-deterministic ones, i.e., every enforcing family defined by a deterministic enforcing set is – by definition – also defined by a non-deterministic enforcing set, but not the other way around.

To explain this, consider the enforcing set $\mathcal{E} = \{(\varnothing, \{a,b\})\}$, that is non-deterministic and for which $\mathcal{L}(\mathcal{E}) = \{L \mid L \text{ contains } a \text{ or } b\}$ (where $a \neq b$). Since $\varnothing \notin \mathcal{L}(\mathcal{E})$, the family $\mathcal{L}(\mathcal{D})$ of languages satisfying an equivalent deterministic enforcing set $\mathcal{D}$ should also not contain $\varnothing$, which means that $\mathcal{D}$ should contain at least one enforcer of the form $(\varnothing, \{z\})$ for some word z. Now, if $z \neq a$ and $z \neq b$, then $\mathcal{L}(\mathcal{D})$ does not contain the languages containing a or b but not z. Moreover, if $z = a$, then none of the languages containing b but not a satisfies $\mathcal{D}$, and similarly for the case that $z = b$. Hence $\mathcal{L}(\mathcal{D}) \neq \mathcal{L}(\mathcal{E})$ for every deterministic enforcing set $\mathcal{D}$.

Another way to argue that $\mathcal{E}$ cannot be equivalent to $\mathcal{D}$ is the observation that $\mathcal{L}(\mathcal{E})$ contains disjoint languages, whereas $\mathcal{L}(\mathcal{D})$ cannot contain disjoint languages because of $(\varnothing, \{z\})$.

The non-deterministic enforcing set $\mathcal{E}$ from the previous two paragraphs is rather special because the only enforcer is 'brute', i.e., of the form $(\varnothing, Y)$. To demonstrate that this is not the only reason why deterministic enforcing sets are less powerful than non-deterministic ones, in the proof below we give another non-deterministic enforcing set, that does not have this special property, and we show that it also does not have a deterministic equivalent.

Theorem 11.15 *There are enforcing sets without brute enforcers for which there is no equivalent deterministic enforcing set.*

Proof. Take the non-deterministic enforcing set $\mathcal{E} = \{(\{a\}, \{b,c\})\}$ (where a, b and c are different words), and observe $\mathcal{L}(\mathcal{E}) \cap \mathcal{P}(\{a,b,c\}) = \{\varnothing, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}$. If there exists a deterministic enforcing set $\mathcal{D}$ with $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{E})$, then it should also hold that $\mathcal{L}(\mathcal{D}) \cap \mathcal{P}(\{a,b,c\}) = \mathcal{L}(\mathcal{E}) \cap \mathcal{P}(\{a,b,c\})$. We show that there is no deterministic enforcing set $\mathcal{D}$ such that the latter equation holds.

First, note that for $\mathcal{D}$ we only have to consider enforcers (X,Y) with $X \subseteq \{a,b,c\}$ and $Y = \{a\}$ or $Y = \{b\}$ or $Y = \{c\}$: if $w \in X$ for some $w \neq a,b,c$, then we may discard this enforcer since it is not applicable to any language in $\mathcal{P}(\{a,b,c\})$, whereas if $Y = \{w\}$ for some $w \neq a,b,c$, then application of such an enforcer (if possible) gives a language not in $\mathcal{P}(\{a,b,c\})$.

Second, note that $\mathcal{D}$ cannot contain an enforcer of the form $(\varnothing, Y)$ for any Y, since $\mathcal{E}$ does not contain enforcers of this form. In other words, $\mathcal{L}(\mathcal{E})$ contains the empty language, thus $\mathcal{L}(\mathcal{D})$ should also contain $\varnothing$.

Third, we only have to look at enforcers of the form (X,Y) for which $X \cap Y = \varnothing$ (this is a normal form discussed in the previous subsection).

From the above it follows that there are only nine possible enforcers for the part of $\mathcal{D}$ that should define $\mathcal{L}(\mathcal{E}) \cap \mathcal{P}(\{a,b,c\})$:

$$\begin{array}{ccc} (\{b\},\{a\}) & (\{a\},\{b\}) & (\{a\},\{c\}) \\ (\{c\},\{a\}) & (\{c\},\{b\}) & (\{b\},\{c\}) \\ (\{b,c\},\{a\}) & (\{a,c\},\{b\}) & (\{a,b\},\{c\}) \end{array}$$

For the three enforcers of the form $(X, \{a\})$ we have that the language $\{b, c\}$ does not satisfy them. Hence $\mathcal{D}$ should not contain any of these three enforcers. The six other cases can be shown to be inappropriate in a similar way. Thus the part of $\mathcal{D}$ that accounts for $\mathcal{L}(\mathcal{E}) \cap \mathcal{P}(\{a,b,c\})$ can only be empty, but in that case the language $\{a\}$ satisfies $\mathcal{D}$, which should not be the case. Consequently there is no deterministic enforcing set that is equivalent to $\mathcal{E}$. □

One consequence of an enforcing set being deterministic is that it makes the notions 'finitary' and 'weakly finitary' equivalent (see Subsection 10.2.3).

Theorem 11.16 *Let $\mathcal{E}$ be a deterministic enforcing set.*
Then $\mathcal{E}$ is finitary if and only if $\mathcal{E}$ is weakly finitary.

Proof. We prove that non-finitary deterministic enforcing sets $\mathcal{E}$ cannot be weakly finitary. Assume that, for a certain finite language X, the set $\mathcal{E}(X)$ is infinite, i.e., $\mathcal{E}$ is not finitary. Let K be a language such that $X \vdash_{\mathcal{E}} K$. Since $\mathcal{E}$ is deterministic, K must contain the set $\{y \mid (X, \{y\}) \in \mathcal{E}\}$, which is an infinite set because $\mathcal{E}(X)$ is infinite. Thus K cannot be finite, and $\mathcal{E}$ is not weakly finitary.

Combining the above with Proposition 10.3(1) completes the proof of the result. □

11.3 Summary

We summarize the more important results of this chapter. Both for forbidding sets and for enforcing sets, we have shown how to remove redundancy from these sets, by proving a series of normal forms. In the case of forbidding sets we could even prove that for each forbidding set a unique redundancy free equivalent can be constructed.

We demonstrated that these normal forms cannot always yield a *finite* equivalent, i.e., we have proved that infinite forbidding and enforcing sets are strictly more powerful than finite ones.

Furthermore, we have shown that non-determinism is an essential feature of enforcing sets, in the sense that there are non-deterministic enforcing sets for which there is no deterministic equivalent.

Chapter 12

Sequences of languages in forbidding-enforcing families

A number of results concerning fe systems deal with the fact that for an infinite ascending sequence of languages satisfying certain properties, their union satisfies the same, or closely related, properties. We generalise some of these results, mainly by lifting them to converging sequences of languages (in the topological sense). Furthermore, we discuss in detail the importance of (sequences of) finite languages in forbidding and enforcing families.

In order to simplify notation, throughout this chapter we consider an arbitrary but fixed alphabet Σ, i.e. every word, language or family of languages is over Σ, unless clear otherwise.

12.1 Converging sequences of languages

By $\langle K_i \rangle_{i \in \mathbb{N}}$ we denote the infinite sequence $K_1, K_2, \ldots$ of languages. We define the *distance* between two languages K and L as follows:

$$d(K, L) = \begin{cases} 0 & \text{if } K = L \\ 2^{-\min\{|x| \ \mid \ x \in K \triangle L\}} & \text{otherwise} \end{cases}$$

where $K \triangle L = (K - L) \cup (L - K)$ is the symmetric difference of K and L. It is fairly easy to verify that d satisfies the usual requirements for (ultrametric) distances ([BV96]):

1. $d(K, L) = 0$ if and only if $K = L$,
2. $d(K, L) = d(L, K)$, and
3. $d(K, L) \leq \max\{d(K, M), d(M, L)\}$

where K, L, M are languages.

For $\ell \geq 0$ we have, by definition, $d(K, L) < 2^{-\ell}$ if and only if $K|_{\leq \ell} = L|_{\leq \ell}$. In other words: the longer the shortest word that 'separates' K from L, the smaller the distance between K and L.

Let us now consider the usual notion of *convergence*: an infinite sequence $\langle K_i \rangle_{i \in \mathbb{N}}$ of languages converges to a language K if $d(K_i, K) \to 0$ when $i \to \infty$. Hence the distance between elements of the sequence $\langle K_i \rangle_{i \in \mathbb{N}}$ and its limit K will become arbitrarily small. As this distance is measured in terms of the length of the shortest word in the symmetric difference, we can rephrase the notion of convergence as follows.

Definition 12.1 An infinite sequence of languages $\langle K_i \rangle_{i \in \mathbb{N}}$ *converges* to a language K, denoted $\langle K_i \rangle_{i \in \mathbb{N}} \to K$, if, for each $\ell \geq 0$, there is an $m \geq 1$ such that $K_n|_{\leq \ell} = K|_{\leq \ell}$ for every $n \geq m$. □

The following easy observation is a frequently used 'technical tool' in investigating converging sequences of languages. Assume that $\langle K_i \rangle_{i \in \mathbb{N}} \to K$. If X is a *finite* language such that $X \subseteq K$, then there is an $m \geq 1$ such that $X \subseteq K_n$ for every $n \geq m$.

Example 12.1

1. $\langle \{a^n\} \rangle_{n \in \mathbb{N}} \to \varnothing$, since, for every $\ell \geq 0$, $\{a^n\}|_{\leq \ell} = \varnothing = \varnothing|_{\leq \ell}$ for all $n \geq \ell + 1$.
2. Analogously, $\langle \Sigma^*|_{>n} \rangle_{n \in \mathbb{N}} \to \varnothing$.
3. The ascending sequence of languages $\langle \{a^1, \ldots, a^n\} \rangle_{n \in \mathbb{N}}$ converges to a^+, since $\{a^1, \ldots, a^\ell\}|_{\leq \ell} = a^+|_{\leq \ell}$ for all $\ell \geq 0$.
4. Analogously, $\langle \{a^1, \ldots, a^n, b^{n+1}\} \rangle_{n \in \mathbb{N}} \to a^+$. Note, however, that this is not an ascending sequence.

□

The framework of metric spaces has been proposed to deal with the semantics of recursion (and with infinite computations in general) by Nivat ([Niv79], see also [BV96]).

12.2 Forbidding-enforcing families are closed sets

We repeat Proposition 10.1(3), which is very useful in analysing families of consistent languages.

Proposition 12.1 *Let $\mathcal{F}$ be a forbidding set, and let $\langle K_i \rangle_{i \in \mathbb{N}}$ be an ascending sequence of languages. If K_i $\underline{con}$ $\mathcal{F}$ for all $i \geq 1$, then $(\bigcup_i K_i)$ $\underline{con}$ $\mathcal{F}$.*

The following lemma is a generalisation of Proposition 12.1, in the sense that we do not require a sequence of ascending languages, the union of which equals a certain language K, but rather a sequence of languages converging to K. Note that if $\langle K_i \rangle_{i \in \mathbb{N}}$ is an ascending sequence, then indeed $\langle K_i \rangle_{i \in \mathbb{N}} \to (\bigcup_i K_i)$.

Lemma 12.1 *Let $\mathcal{F}$ be a forbidding set and K a language. If $\langle K_i \rangle_{i \in \mathbb{N}}$ is an infinite sequence of languages with $\langle K_i \rangle_{i \in \mathbb{N}} \to K$ and $K_i \, \underline{con} \, \mathcal{F}$ for all $i \geq 1$, then $K \, \underline{con} \, \mathcal{F}$.*

Proof. Assume that not $K \, \underline{\text{con}} \, \mathcal{F}$, i.e., there exists an $F \in \mathcal{F}$ such that $F \subseteq \underline{\text{sub}}\,(K)$. Hence $F \subseteq \underline{\text{sub}}\,(X)$ for a finite $X \subseteq K$. Since $\langle K_i \rangle_{i \in \mathbb{N}} \to K$ there is an $m \geq 1$ such that $X \subseteq K_n$ for all $n \geq m$. Thus $F \subseteq \underline{\text{sub}}\,(K_m)$, which contradicts $K_m \, \underline{\text{con}} \, \mathcal{F}$. □

An analogous result holds for enforcing sets.

Lemma 12.2 *Let $\mathcal{E}$ be an enforcing set and K a language. If $\langle K_i \rangle_{i \in \mathbb{N}}$ is an infinite sequence of languages with $\langle K_i \rangle_{i \in \mathbb{N}} \to K$ and $K_i \, \underline{sat} \, \mathcal{E}$ for all $i \geq 1$, then $K \, \underline{sat} \, \mathcal{E}$.*

Proof. Let $(X, Y) \in \mathcal{E}$ and choose $k = \max\{ \, |w| \mid w \in X \cup Y \}$. Since $\langle K_i \rangle_{i \in \mathbb{N}} \to K$ there is an $m \geq 1$ such that $K_m|_{\leq k} = K|_{\leq k}$. We know that $K_m \, \underline{\text{sat}} \, (X, Y)$, hence $X \subseteq K_m$ implies $K_m \cap Y \neq \varnothing$. But $X \subseteq K_m$ if and only if $X \subseteq K$, and $K_m \cap Y = K \cap Y$, since X and Y do not contain words of length greater than k. Thus $K_m \, \underline{\text{sat}} \, (X, Y)$ implies $K \, \underline{\text{sat}} \, (X, Y)$. Consequently, $K \, \underline{\text{sat}} \, (X, Y)$ for every $(X, Y) \in \mathcal{E}$. □

Lemma 12.1, Lemma 12.2 and the fact that $\mathcal{L}(\mathcal{F}, \mathcal{E}) = \mathcal{L}(\mathcal{F}) \cap \mathcal{L}(\mathcal{E})$ directly imply that $\mathcal{L}(\mathcal{F}, \mathcal{E})$ is a *closed set* ([Smy92]), where a family of languages $\mathcal{C}$ is called a closed set if, for each sequence $\langle K_i \rangle_{i \in \mathbb{N}}$ of languages in $\mathcal{C}$, $\langle K_i \rangle_{i \in \mathbb{N}} \to K$ implies that $K \in \mathcal{C}$.

Theorem 12.3 *Let $(\mathcal{F}, \mathcal{E})$ be an fe system. Then $\mathcal{L}(\mathcal{F}, \mathcal{E})$ is a closed set in the topology induced by the metric d.*

12.3 Evolving sequences of languages

The basic computational feature of an fe system is 'evolving through enforcing', determined by the enforcing set of the system. It is formalized through the extension relation, see Definition 10.3.

In this section we discuss sequences of languages in which each language is an $\mathcal{E}$-extension of its direct predecessor, for an enforcing set $\mathcal{E}$. In other words,

each language in such a sequence evolves into its direct successor according to the $\mathcal{E}$-extension relation $\vdash_{\mathcal{E}}$.

We recall an important result on infinite ascending evolving sequences, that says that for such a sequence the union of the languages occurring in it satisfies $\mathcal{E}$ (Proposition 10.2).

Note that, unlike in [EH$^+$00], here we do not assume that $K \vdash_{\mathcal{E}} L$ implies $K \subseteq L$ – the latter is also the setup in [ER]. (This assumption was made in [EH$^+$00] because we considered there ascending sequences of languages.) Therefore, $\vdash_{\mathcal{E}}$ in general is not a transitive relation, although for ascending sequences it *is* transitive. For example, take $\mathcal{E} = \{ (\{a\},\{b\}),\ (\{a\},\{c\}),\ (\{b\},\{a\}) \}$, then $\{a\} \vdash_{\mathcal{E}} \{b,c\} \vdash_{\mathcal{E}} \{a\}$, but not $\{a\} \vdash_{\mathcal{E}} \{a\}$. Because of this non-transitivity, $K_i \vdash_{\mathcal{E}} K_{i+1}$ seems to be a rather local property, even if it holds for every $i \geq 1$ in some sequence of languages $\langle K_i \rangle_{i \in \mathbb{N}}$.

Still, the $\mathcal{E}$-extension relation turns out to be strong enough to yield the following result.

Theorem 12.4 *Let $\mathcal{E}$ be an enforcing set and K a language. If $\langle K_i \rangle_{i \in \mathbb{N}}$ is an infinite sequence of languages with $\langle K_i \rangle_{i \in \mathbb{N}} \to K$ and $K_i \vdash_{\mathcal{E}} K_{i+1}$ for all $i \geq 1$, then K $\underline{sat}$ $\mathcal{E}$.*

Proof. Assume that $(X,Y) \in \mathcal{E}$ with $X \subseteq K$. Since X and Y are finite, we can define $k = \max\{\, |w| \mid w \in X \cup Y\}$. Because $\langle K_i \rangle_{i \in \mathbb{N}} \to K$ there is an $\ell \geq 1$ such that $K_j|_{\leq k} = K|_{\leq k}$ for all $j \geq \ell$, hence $X \subseteq K_j$ for all $j \geq \ell$.

Then $Y \cap K_{\ell+1} \neq \varnothing$ since $X \subseteq K_\ell$ and $K_\ell \vdash_{\mathcal{E}} K_{\ell+1}$. Because $\ell + 1 \geq \ell$ we have $K_{\ell+1}|_{\leq k} = K|_{\leq k}$, hence $Y \cap K_{\ell+1} = Y \cap K \neq \varnothing$. □

12.4 The importance of finite languages

In this section we consider (possibly infinite) sequences of (specific) *finite* languages – the underlying observation is that every language K can be written in the form $K = \bigcup_{n \geq 0} K|_{\leq n}$. Note that $\langle K|_{\leq n} \rangle_{n \geq 0}$ converges to K.

Proposition 10.1 (2) and (3) together yield the following result, which states that consistence of certain specific finite parts of a language K is necessary and sufficient to ensure the consistence of K itself.

Theorem 12.5 *K $\underline{con}$ $\mathcal{F}$ if and only if $K|_{\leq n}$ $\underline{con}$ $\mathcal{F}$ for every $n \geq 0$.*

For enforcing sets the situation is slightly different, because enforcers force the presence of certain *words*, whereas forbidders prevent the occurrence of certain sets of *subwords* (see also the remark at the end of Subsection 10.2.1).

We define $\mathcal{E}|_{\leq n}$ to be $\{(X,Y) \in \mathcal{E} \mid |w| \leq n \text{ for all } w \in X \cup Y\}$.

Theorem 12.6 *K $\underline{sat}$ $\mathcal{E}$ if and only if $K|_{\leq n}$ $\underline{sat}$ $\mathcal{E}|_{\leq n}$ for every $n \geq 0$.*

Proof. Assume that $K\,\underline{\text{sat}}\,\mathcal{E}$. Take an arbitrary $n \geq 0$, and consider $(X, Y) \in \mathcal{E}|_{\leq n}$. Obviously, $X \subseteq K$ if and only if $X \subseteq K|_{\leq n}$, and $Y \cap K = Y \cap K|_{\leq n}$. Hence $K\,\underline{\text{sat}}\,\mathcal{E}$ implies $K|_{\leq n}\,\underline{\text{sat}}\,\mathcal{E}|_{\leq n}$.

On the other hand, if $K|_{\leq n}\,\underline{\text{sat}}\,\mathcal{E}|_{\leq n}$ for every $n \geq 0$, then it also holds that $(K|_{\leq n} \cup \Sigma^*|_{>n})\,\underline{\text{sat}}\,\mathcal{E}$ for every $n \geq 0$. Since $\langle K|_{\leq n} \cup \Sigma^*|_{>n}\rangle_{n\geq 0} \to K$, we can apply Lemma 12.2 to obtain $K\,\underline{\text{sat}}\,\mathcal{E}$. □

Note that the sequence $\langle K|_{\leq n} \cup \Sigma^*|_{>n}\rangle_{n\geq 0}$ used in the proof above is descending rather than ascending.

The membership of a language in an fe family is determined by its finite portions only; this is an elementary consequence of the fact that these families are closed sets.

Theorem 12.7 *Let $\mathcal{K}$ be an fe family, and K a language. If, for each $n \geq 0$, there is an $L \in \mathcal{K}$ with $K|_{\leq n} = L|_{\leq n}$, then $K \in \mathcal{K}$.*

Proof. Let $\mathcal{K} = \mathcal{L}(\mathcal{F}, \mathcal{E})$, and let, for every $n \geq 0$, $L_n \in \mathcal{K}$ be a language with $L_n|_{\leq n} = K|_{\leq n}$. Since $\langle L_n\rangle_{n\geq 0} \to K$, Theorem 12.3 gives $K \in \mathcal{K}$. □

We state now two interesting corollaries of Theorem 12.7.

Corollary 12.8 *Let $\mathcal{K}_1$ and $\mathcal{K}_2$ be fe families.*
If, for all $n \geq 0$, $\{K|_{\leq n} \mid K \in \mathcal{K}_1\} = \{K|_{\leq n} \mid K \in \mathcal{K}_2\}$, then $\mathcal{K}_1 = \mathcal{K}_2$.

Proof. Let $K \in \mathcal{K}_1$. Then, according to the condition in the corollary, for every $n \geq 0$ there is an $L_n \in \mathcal{K}_2$ with $K|_{\leq n} = L_n|_{\leq n}$. Hence by Theorem 12.7 we have $K \in \mathcal{K}_2$, and thus $\mathcal{K}_1 \subseteq \mathcal{K}_2$. Obviously, $\mathcal{K}_2 \subseteq \mathcal{K}_1$ can be proved analogously. □

It is perhaps superfluous to remark that the implication of Corollary 12.8 does not hold for arbitrary $\mathcal{K}_1$ and $\mathcal{K}_2$, as can be seen by letting $\mathcal{K}_1$ and $\mathcal{K}_2$ be the respective families of finite and infinite languages over a fixed alphabet.

Suppose that an fe family $\mathcal{K}$ contains all finite languages over a certain alphabet Δ. Then $K|_{\leq n} \in \mathcal{K}$ for each language K over this alphabet and each $n \geq 0$, hence according to the theorem $K \in \mathcal{K}$ for all K.

Corollary 12.9 *Let $\mathcal{K}$ be an fe family and $\Delta \subseteq \Sigma$. If $\mathcal{K}$ contains all finite languages over Δ, then it contains all languages over Δ.*

This second corollary also follows directly from the definitions of forbidding and enforcing, which is seen as follows. Let $\mathcal{K} = \mathcal{L}(\mathcal{F}, \mathcal{E})$.

If every finite $L \subseteq \Delta^*$ is consistent with $\mathcal{F}$, then every forbidder $F \subseteq \Delta^*$ in $\mathcal{F}$ should be consistent with $\mathcal{F}$ as well. Hence these forbidders cannot exist in $\mathcal{F}$. Consequently, every $F \in \mathcal{F}$ contains at least one symbol from $\Sigma - \Delta$ and thus $M\,\underline{\text{con}}\,\mathcal{F}$ holds for all $M \subseteq \Delta^*$.

Furthermore, enforcers (X, Y) such that X contains symbols from $\Sigma - \Delta$ are obviously not applicable to languages over Δ. Now let $(X, Y) \in \mathcal{E}$ with $X \subseteq \Delta^*$. Then $X \ \underline{\text{sat}}\ (X, Y)$ should hold, since every finite language over Δ satisfies $\mathcal{E}$. In other words, it should be the case that $X \cap Y \neq \varnothing$. Obviously, such an enforcer is satisfied by every language. This ends our alternative explanation of Corollary 12.9.

We consider now separately families of languages defined by forbidding sets and families of languages defined by enforcing sets.

For forbidding families the finite languages are most crucial: in fact, they determine the family.

Theorem 12.10 *For all forbidding sets $\mathcal{F}_1$ and $\mathcal{F}_2$ the following holds:* $\mathcal{L}(\mathcal{F}_1) = \mathcal{L}(\mathcal{F}_2)$ *if and only if* $\mathcal{L}_{\text{fin}}(\mathcal{F}_1) = \mathcal{L}_{\text{fin}}(\mathcal{F}_2)$.

Proof. It is clear that $\mathcal{L}(\mathcal{F}_1) = \mathcal{L}(\mathcal{F}_2)$ implies $\mathcal{L}_{\text{fin}}(\mathcal{F}_1) = \mathcal{L}_{\text{fin}}(\mathcal{F}_2)$.

Now let $L \ \underline{\text{con}}\ \mathcal{F}_1$. Then $L|_{\leq i} \ \underline{\text{con}}\ \mathcal{F}_1$ for all $i \geq 0$, and because $\mathcal{L}_{\text{fin}}(\mathcal{F}_1) = \mathcal{L}_{\text{fin}}(\mathcal{F}_2)$ also $L|_{\leq i} \ \underline{\text{con}}\ \mathcal{F}_2$ for all $i \geq 0$. Now Theorem 12.5 gives $L \ \underline{\text{con}}\ \mathcal{F}_2$. Analogously it can be proved that $L \ \underline{\text{con}}\ \mathcal{F}_2$ implies $L \ \underline{\text{con}}\ \mathcal{F}_1$. □

Unlike for forbidding families, finite languages are not particularly important for families of satisfying languages. This can be shown as follows.

Example 12.2 Let K be a language, and let $\langle w_i \rangle_{i \in \mathbb{N}}$ be an arbitrary but fixed ordering of the words of K. Similarly, let $\langle v_i \rangle_{i \in \mathbb{N}}$ be an arbitrary but fixed ordering of the elements of $\Sigma^* - K$. Now consider the enforcing set $\mathcal{E}_K = \{\ (\varnothing, \{w_1\})\ \} \cup \{\ (\{w_i\}, \{w_{i+1}\})\ \mid i \geq 1\} \cup \{\ (\{v_i\}, \{v_1\}),\ (\{v_i\}, \{v_{i+1}\})\ \mid i \geq 1\}$, then $\mathcal{L}(\mathcal{E}_K) = \{K, \Sigma^*\}$.

Now clearly for every finite language K we have $\mathcal{L}_{\text{fin}}(\mathcal{E}_K) = \{K\}$, whereas for infinite K we have $\mathcal{L}_{\text{fin}}(\mathcal{E}_K) = \varnothing$. Hence for any two different infinite languages K and K' we have $\mathcal{L}_{\text{fin}}(\mathcal{E}_K) = \mathcal{L}_{\text{fin}}(\mathcal{E}_{K'}) = \varnothing$, whereas $\mathcal{L}(\mathcal{E}_K) \neq \mathcal{L}(\mathcal{E}_{K'})$. □

The results of this section point out a crucial difference between languages defined by fe systems and languages defined by classical grammars, such as, e.g., Chomsky grammars. As demonstrated above, finite languages are very important for fe systems – they in fact determine fe language families. On the other hand, finite languages are irrelevant for Chomsky grammars: if a language L is of type X (regular, context-free, context-sensitive, ...), then so are $L \cup F$ and $L - F$, for every finite language F.

12.5 Summary

We have extended Proposition 10.1(3) – that states that if every language in an ascending sequence of languages is consistent with the forbidding set $\mathcal{F}$,

then the union of these languages is consistent with $\mathcal{F}$ as well – to the more general notion of converging sequences of languages in forbidding, enforcing or forbidding-enforcing families.

A similar result was known for infinite ascending evolving sequences of languages (Proposition 10.2). We have extended this result to infinite converging evolving sequences.

Finally, we have illustrated that finite (parts of) languages are characteristic for fe families, by showing that the membership of a language in an fe family is determined by specific finite subsets of that language, that if an fe family contains all finite languages over a certain alphabet, then it contains all languages over this alphabet, and that two forbidding sets are equivalent if and only if they define the same finite languages.

Bibliography

[Adl94] L.M. Adleman. Molecular computation of solutions to combinatorial problems, *Science* 226:1021–1024, 1994.

[Aho90] A.V. Aho. Algorithms for finding patterns in strings, *Handbook of theoretical computer science* (J. van Leeuwen, ed.) volume A:255–300, Elsevier Science Publishers, 1990.

[Amo97] M. Amos. DNA *computation*, Ph.D. thesis, Department of Computer Science, University of Warwick, UK, 1997.

[AU70] A.V. Aho, J.D. Ullman. A characterization of two-way deterministic classes of languages, *Journal of Computer and System Sciences* 4(6):523–538, 1970.

[BV96] J. de Bakker, E. de Vink. *Control flow semantics*, MIT Press, Cambridge, MA, 1996.

[BZ99] P. Bonizzoni, R. Zizza. Deciding whether a regular language is a splicing language, Technical Report, Dipartimento di Scienze dell' Informazione, 2000.

[CH91] K. Culik II, T. Harju. Splicing semigroups of dominoes and DNA, *Discrete Applied Mathematics* 31:261–277, 1991.

[DHV01] J. Dassen, H.J. Hoogeboom, N. van Vugt. A characterization of non-iterated splicing with regular rules, *Where mathematics, computer science, linguistics and biology meet* (C. Martín-Vide, V. Mitrana, eds.) 319-327, Kluwer, 2001.

[Dic13] L.E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors, *American Journal of Mathematics* 35:413–422, 1913.

[EH+00] A. Ehrenfeucht, H.J. Hoogeboom, G. Rozenberg, N. van Vugt. Forbidding and enforcing, DNA *based computers V* (E. Winfree, D.K. Gifford, eds.), DIMACS Series in Discrete Mathematics 54 (1999 proceedings), 2000.

[EH+01] A. Ehrenfeucht, H.J. Hoogeboom, G. Rozenberg, N. van Vugt. Sequences of languages in forbidding-enforcing families, *Soft Computing* 5(2):121–125, 2001.

[ER] A. Ehrenfeucht, G. Rozenberg. Forbidding-enforcing systems, manuscript.

[FMR68] P.C. Fischer, A.R. Meyer, A.L. Rosenberg. Counter machines and counter languages, *Mathematical Systems Theory* 2:265–283, 1968.

[FP+98] R. Freund, Gh. Păun, G. Rozenberg, A. Salomaa. Bidirectional sticker systems, *Third Annual Pacific Conference on Biocomputing*, Hawaii, 1998 (R.B. Altman, A.K. Dunker, L. Hunter, T.E. Klein, eds.), 535–546, World Scientific, Singapore, 1998.

[FS97] H. Fernau, R. Stiebe. Regulation by valences, *Mathematical Foundations of Computer Science 1997*, Lecture Notes in Computer Science 1295 (Igor Prívara, Peter Ruzicka, eds.), 239–248, Springer-Verlag, 1997.

[FS00] H. Fernau, R. Stiebe. Sequential grammars and automata with valences, technical report WSI-2000-25, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 2000.

[GG69] S. Ginsburg, S.A. Greibach. Abstract families of languages, *Memoirs of the American Mathematical Society* 87:1–32, 1969.

[Gin75] S. Ginsburg. *Algebraic and automata-theoretic properties of formal languages*, North-Holland/American Elsevier, 1975.

[GJ79] M.R. Garey, D.S. Johnson. *Computers and intractability, a guide to the theory of NP-completeness*, Bell Telephone Laboratories, 1979.

[Goo99] E. Goode Laun. *Constants and splicing systems*, Dissertation, State University of New York at Binghamton, 1999.

[Gre78] S.A. Greibach. Remarks on blind and partially blind one-way multicounter machines, *Theoretical Computer Science* 7:311–324, 1978.

[Gre79] S.A. Greibach. Linearity is polynomially decidable for realtime pushdown store automata, *Information and Control* 42(1):27–37,1979.

[Har67] J. Hartmanis. Context-free languages and Turing machine computations, *Mathematical Aspects of Computer Science* 19:42–51, Proc. Symp. Applied Mathematics (J.T. Schwartz, ed.), American Mathematical Society, 1967.

[Hea87] T. Head. Formal language theory and DNA: an analysis of the generative capacity of specific recombinant behaviors, *Bulletin of Mathematical Biology* 49(6):737–759, 1987.

[Hea98] T. Head. Splicing languages generated with one-sided context, *Computing with bio-molecules: theory and experiments* (Gh. Păun, ed.), 269–282, Springer-Verlag, Singapore, 1998.

[HH99] V. Halava, T. Harju. Languages accepted by integer weighted finite automata, *Jewels are forever* (J. Karhumäki, H. Maurer, Gh. Păun, G. Rozenberg, eds.), 123–134, Springer-Verlag, 1999.

[Hoo02] H.J. Hoogeboom. Context-free valence grammars – revisited, *Developments in Language Theory 2001* (W. Kuich, G. Rozenberg, A. Salomaa, eds.), Lecture Notes in Computer Science 2295:293-303, Springer-Verlag, 2002.

[HPP97] T. Head, Gh. Păun, D. Pixton. Language theory and molecular genetics: generative mechanisms suggested by DNA recombination, *Handbook of formal languages* (G. Rozenberg, A. Salomaa, eds.), volume 2: 295–360, Springer-Verlag, 1997.

[HU79] J.E. Hopcroft, J.D. Ullman. *Introduction to automata theory, languages, and computation*, Addison-Wesley, 1979.

[HV98] H.J. Hoogeboom, N. van Vugt. The power of H systems: does representation matter? *Computing with bio-molecules: theory and experiments* (Gh. Păun, ed.), 255–268, Springer-Verlag, Singapore, 1998.

[HV00] H.J. Hoogeboom, N. van Vugt. Fair sticker languages, *Acta Informatica* 37: 213–225, 2000.

[HV02] H.J. Hoogeboom, N. van Vugt. Upper bounds for restricted splicing, *Formal and natural computing - essays dedicated to Grzegorz Rozenberg* (W. Brauer, H. Ehrig, J. Karhumäki, A. Salomaa, eds.), Lecture Notes in Computer Science 2300:361–375, Springer Verlag, 2002.

[KP+98] L. Kari, Gh. Păun, G. Rozenberg, A. Salomaa, S. Yu. DNA computing, sticker systems, and universality, *Acta Informatica* 35:401–420, 1998.

[KPS96] L. Kari, Gh. Păun, A. Salomaa. The power of restricted splicing with rules from a regular language, *Journal of Universal Computer Science* 2(4):224-240, 1996.

[Lat79] M. Latteux. Cônes rationnels commutatifs, *Journal of Computer and System Sciences* 18:307–333, 1979.

[LLR85] M. Latteux, B. Leguy, B. Ratoandromanana. The family of one-counter languages is closed under quotient, *Acta Informatica* 22:579–588, 1985.

[NEB] New England BioLabs Catalog 1998/1999.

[Niv79] M. Nivat. Infinite words, infinite trees, infinite computations, *Foundations of computer science III, part 2 (Languages, logic, semantics)* (J. W. de Bakker, J. van Leeuwen, eds.), 3–52, Mathematical Centre Amsterdam, 1979.

[Pău80] Gh. Păun. A new generative device: valence grammars, *Revue Roumaine de Mathématiques Pures et Appliquées* 25(6):911–924, 1980.

[Pău96a] Gh. Păun. On the splicing operation, *Discrete Applied Mathematics* 70:57–79, 1996.

[Pău96b] Gh. Păun. Regular extended H systems are computationally universal, *Journal of Automata, Languages and Combinatorics* 1(1):27–36, 1996.

[Pău98] Gh. Păun (ed.). *Computing with bio-molecules: theory and experiments*, Springer-Verlag, Singapore, 1998.

[Pix95] D. Pixton. Linear and circular splicing systems, *Proceedings of the 1st International Symposium on Intelligence in Neural and Biological Systems*, 38–45, IEEE, 1995.

[Pix96] D. Pixton. Regularity of splicing languages, *Discrete Applied Mathematics* 69:101-124, 1996.

[PR98] Gh. Păun, G. Rozenberg. Sticker systems, *Theoretical Computer Science* 204:183–203, 1998.

[PRS96a] Gh. Păun, G. Rozenberg, A. Salomaa. Computing by splicing, *Theoretical Computer Science* 168:321–336, 1996.

[PRS96b] Gh. Păun, G. Rozenberg, A. Salomaa. Restricted use of the splicing operation, *International Journal of Computer Mathematics* 60:17–32, 1996.

[PRS98] Gh. Păun, G. Rozenberg, A. Salomaa. DNA *computing. New computing paradigms*, Springer-Verlag, 1998.

[RS97] G. Rozenberg, A. Salomaa (eds.). *Handbook of formal languages*, Springer-Verlag, 1997.

[RW$^+$98] S. Roweis, E. Winfree, R. Burgoyne, N.V. Chelyapov, M.F. Goodman, P.W.K. Rothemund, L.M. Adleman. A sticker based model for DNA computation, DNA *based computers II*, DIMACS Series in Discrete Mathematics 44 (1996 proceedings), 1998.

[RW99] H. Rubin, D.H. Wood. DNA *based computers III*, DIMACS Series in Discrete Mathematics 48 (1997 proceedings), 1999.

[Smy92] M. B. Smyth. Topology, *Handbook of logic in computer science (Background: mathematical structures)* (S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, eds.), 641–761, Oxford Science Publications, 1992.

[Win98] E. Winfree. *Algorithmic self-assembly of* DNA, Ph.D. thesis, California Institute of Technology, Pasadena, California, 1998.

[WYS98] E. Winfree, X. Yang, N.C. Seeman. Universal computation via self-assembly of DNA: some theory and experiments, DNA *based computers II*, DIMACS Series in Discrete Mathematics 44 (1996 proceedings), 1998.

Samenvatting

Gemotiveerd door een experiment van Adleman, dat gepubliceerd is in *Science*, zijn informatici tegenwoordig geïnteresseerd in de mogelijkheden die DNA kan bieden om complexe berekeningen te maken. We hebben onderzoek gedaan naar drie modellen die drie verschillende processen beschrijven die met (DNA-) moleculen te maken hebben. Alledrie de modellen zijn opgesteld binnen de formele-talentheorie, wat wil zeggen dat we moleculen weergeven als rijtjes letters in plaats van als een complexe biochemische structuur. Zo'n rijtje letters wordt een string genoemd, en een verzameling van strings heet een (formele) taal. Een taal kan eindig of oneindig veel strings bevatten, en iedere taal heeft een bepaalde moeilijkheidsgraad (behoort tot een bepaalde talenfamilie): de eindige talen zijn eenvoudiger dan de oneindige, en binnen de oneindige talen bestaan ook weer gradaties. Ook als een taal oneindig veel strings bevat, is er vaak een eindige beschrijving van te geven. De moeilijkheidsgraad van de taal wordt bepaald door het type beschrijving.

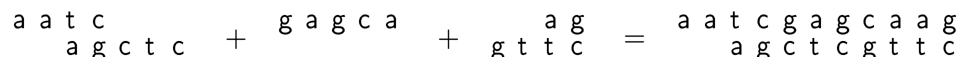
DNA

We leggen nu eerst kort uit wat DNA is en welke eigenschappen ervan door de eerste twee modellen die we bekeken hebben beschreven worden.

De bouwstenen van DNA zijn vier basen (deelmoleculen), die worden aangeduid met de letters a, c, g en t. Heel eenvoudig gezegd bestaat een (dubbelstrengs) DNA-molecuul uit twee rijtjes basen, die *complementair* zijn: een a plakt altijd op een t en andersom, en een c plakt altijd op een g en andersom. Schematisch kan zo'n dubbelstrengs DNA-molecuul als volgt worden weergegeven:

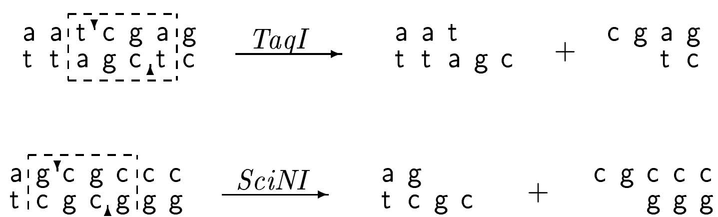
```
a a t c g a g
t t a g c t c
```

Het kan ook voorkomen dat de bovenste of de onderste streng ontbreekt – dan heet het een enkelstrengs molecuul – of dat er aan het linker- en/of rechteruiteinde een enkelstrengs stuk uitsteekt (boven of onder). In het laatste geval noemen we het molecuul gedeeltelijk dubbelstrengs. Enkelstrengs moleculen en/of uitsteeksels kunnen aan elkaar plakken en zo een langer molecuul vormen, mits de enkelstrengs stukken complementair zijn zoals hierboven beschreven:



Dit aan elkaar plakken van enkelstrengs stukken DNA gebeurt spontaan (d.w.z. vanzelf), en is een belangrijk onderdeel van Adlemans experiment. Het tweede model dat wij bekijken – stickersystemen – is een formalisatie van dit spontaan aan elkaar plakken.

Het eerste model dat we behandelen – splicingsystemen – is gebaseerd op een andere eigenschap van DNA: de moleculen kunnen worden doorgeknipt door *restrictie-enzymen*. Een restrictie-enzym zoekt altijd een bepaald rijtje basen op in een DNA-molecuul, en knipt dan het molecuul door op een vaste plek binnen dat rijtje, of juist een zeker aantal plekken verderop. Dit doorknippen gebeurt niet noodzakelijk recht, maar soms met uitsteeksels zoals hieronder aangegeven voor de enzymen *TaqI* en *SciNI*:



De zo ontstane moleculen met uitsteeksels kunnen dan weer aan elkaar of aan andere geschikte uitsteeksels plakken.

We zullen nu één voor één de drie modellen die we onderzocht hebben beschrijven.

Splicing

Splicingsystemen beschrijven de gevolgen van het in twee stukken knippen van DNA-moleculen door restrictie-enzymen en het vervolgens spontaan weer aan elkaar plakken van de stukken. (Het Engelse werkwoord 'to splice' betekent 'verbinden' of 'koppelen'.) Omdat zo'n stuk van een DNA-molecuul niet alleen aan zijn oorspronkelijke wederhelft gekoppeld kan worden, maar ook aan een stuk dat van een ander molecuul is geknipt, ontstaat op deze manier uit de beginverzameling moleculen een nieuwe verzameling.

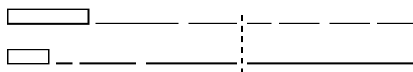
Zowel de DNA-moleculen zelf als het knipgedrag van restrictie-enzymen kunnen worden weergegeven als strings. Hieruit volgt dat een verzameling moleculen gerepresenteerd kan worden door een taal, en een verzameling enzymen ook. Een splicingsysteem bestaat uit een begintaal (de moleculen) en een regeltaal (de enzymen), en produceert zelf weer twee soorten talen: de een bestaat uit alle strings die ontstaan door regels uit de regeltaal één keer toe te passen op strings uit de begintaal, de ander uit alle strings die je kunt krijgen

door herhaald regels toe te passen op beginstrings, maar ook op strings die tijdens dit proces ontstaan zijn. In beide gevallen kun je bekijken welke invloed de moeilijkheidsgraden van begin- en regeltaal hebben op de moeilijkheidsgraad van de resulterende taal.

Ons onderzoek naar splicingsystemen gaat over drie verschillende aspecten ervan. We zijn begonnen met het bekijken van andere stringrepresentaties van enzymen dan de representatie die standaard in de literatuur gebruikt wordt. Daaruit is gebleken dat het in de meeste (maar niet alle) gevallen niet uitmaakt welke van de onderzochte representaties je gebruikt. Daarna hebben we aangetoond dat in een aantal gevallen de moeilijkheidsgraad van regeltalen verlaagd kan worden van (de eenvoudigste vorm van) oneindig naar eindig. Tenslotte hebben we bekeken wat de hoogste moeilijkheidsgraden zijn die de resulterende talen kunnen bereiken als je extra eisen gaat opleggen aan het toepassen van regels op strings, zoals bijvoorbeeld ‘de resulterende string moet altijd langer zijn dan de beide beginstrings’.

Stickers

Stickersystemen zijn een formalisatie van het spontaan aan elkaar plakken van complementaire stukjes DNA. De stickersystemen die wij bekeken hebben bestaan uit een eindig aantal gedeeltelijk dubbelstrengs beginmoleculen en een eindig aantal stickers (enkelstrengs moleculen), die verdeeld zijn in twee groepen: onder- en bovenstickers. Een berekening van een stickersysteem bestaat uit een beginmolecuul met aan de rechterkant eindig veel boven- en onderstickers eraan vastgeplakt, zodat een volledig dubbelstrengs molecuul ontstaat; de bovenstickers mogen alleen in de bovenste streng gebruikt worden, de onderstickers alleen in de onderste streng. Behalve aan complementaire uitsteeksels mogen stickers hier ook aan (de rechterkant van) een molecuul zonder uitsteeksel plakken. Een berekening kan er dan schematisch als volgt uitzien (de twee blokjes zijn de boven- en onderstreng van het beginmolecuul, de horizontale lijntjes stellen de stickers voor, en de verticale stippellijn geeft een positie aan waar een boven- en een ondersticker tegen een volledig dubbelstrengs molecuul zijn geplakt in plaats van aan een complementair uitsteeksel):



Een berekening wordt *fair* genoemd als er evenveel boven- als onderstickers in gebruikt worden, en *primitief* als iedere sticker aan een uitsteeksel geplakt is. De berekening hierboven is dus niet fair en niet primitief.

De (gewone sticker-) taal van een stickersysteem bestaat uit alle strings die bovenstrengen van berekeningen van dat systeem representeren; de stickertaal heet fair (primitief) als je alleen naar faire (primitieve) berekeningen kijkt.

Ons onderzoek op het gebied van stickersystemen bestaat uit twee delen: het vinden van een goede (d.w.z. zo precies mogelijke) bovengrens voor de moeilijkheidsgraad van faire stickertalen, en het zoeken naar verschillen en overeenkomsten tussen verzamelingen van gewone, faire, primitieve en primitief faire stickertalen. Uit dit laatste onderzoek is onder andere gebleken dat de gewone stickertaal (van een willekeurig stickersysteem) ook gegenereerd kan worden met alleen maar faire berekeningen (door een ander stickersysteem), en ook met alleen maar primitieve of alleen maar primitief faire berekeningen.

Forbidding en enforcing

Het derde model dat we onderzocht hebben beschrijft een heel ander soort moleculaire systemen, die zich binnen bepaalde grenzen vrij kunnen ontwikkelen. In het bewuste model, *forbidding-enforcingsystemen*, wordt de ontwikkeling van het systeem gestuurd door enforcingcondities maar tegelijkertijd beperkt door forbiddingcondities.

Iedere enforcingconditie zegt dat wanneer een bepaald eindig groepje moleculen in het systeem aanwezig is er ook ooit minstens één molecuul uit een ander eindig groepje zal ontstaan (door een reactie tussen de al aanwezige moleculen). Dus de enforcingcondities zorgen ervoor dat er steeds nieuwe moleculen aan het systeem worden toegevoegd. Forbiddingcondities daarentegen beperken de evolutie van het systeem door bepaalde eindige groepjes deelmoleculen te verbieden (het systeem gaat 'dood' als alle deelmoleculen uit zo'n groepje op hetzelfde moment in het systeem voorkomen).

Net als in de andere twee modellen beschrijven we moleculen weer met behulp van strings. In tegenstelling tot splicing- en stickersystemen levert een forbidding-enforcingsysteem niet één enkele taal op, maar een hele verzameling talen: alle talen die voldoen aan de condities.

We hebben onderzocht hoe we het aantal forbidding- en enforcingcondities kunnen terugbrengen zonder de verzameling van talen die eraan voldoen te veranderen. Het bleek dat je soms zelfs een oneindige set condities kunt vervangen door een eindige set, maar lang niet altijd. Naast nog een paar andere typisch formele-talenvraagstukken hebben we ook gekeken naar eigenschappen van reeksen van talen die aan de condities voldoen, zoals reeksen waarin iedere taal een uitbreiding is van zijn voorganger (willekeurig, of juist strikt volgens de enforcingcondities), en convergerende reeksen. Zulke taalreeksen stellen in zekere zin de ontwikkeling van het systeem voor, binnen de grenzen die door de twee typen condities worden aangegeven.

Curriculum vitae

Nikè van Vugt is geboren in Waspik, Noord-Brabant, op 23 januari 1972. Van 1984 tot 1990 bezocht zij het Dr. Mollercollege te Waalwijk, waar zij met lof het Gymnasium B diploma behaalde. Van 1990 tot 1996 studeerde zij Informatica aan de Universiteit Leiden, en slaagde daarnaast voor de propedeuse Franse Taal- en Letterkunde en een aantal vakken van Algemene Taalwetenschap. De laatste twee jaar van haar studie gaf zij ook werkgroepen aan studenten. In 1996 begon zij, nog steeds in Leiden, aan haar promotieonderzoek op het gebied van de Theoretische Informatica (i.h.b. DNA computing), onder begeleiding van prof. dr. G. Rozenberg en dr. H.J. Hoogeboom.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-1

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-2

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-3

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-4

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-5

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-6

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-7

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-8

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-9

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model.* Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing.* Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes.* Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search.* Faculty of Mathematics and Natural Sciences, Univ. Leiden. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection.* Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Schedulere Optimization in Real-Time Distributed Databases.* Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics.* Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems.* Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods.* Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems.* Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems.* Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules.* Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System.* Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars.* Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Progam Construction.* Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic.* Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms.* Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes.* Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols.* Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor.* Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant.* Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs.* Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language.* Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure.* Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication.* Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle.* Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection.* Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences.* Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes.* Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes.* Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking.* Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols.* Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents.* Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ.* Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking.* Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects..* Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity.* Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems..* Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing.* Faculty of Mathematics and Natural Sciences, UL. 2002-07