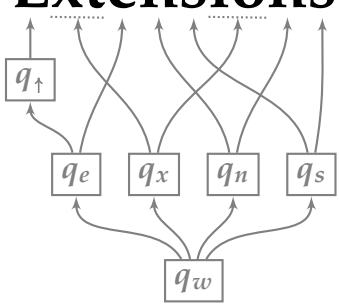
Complexities of Order-Related Formal Language Extensions

Martin Berglund



Department of Computing Science
Umeå 2014

Complexities of Order-Related Formal Language Extensions

Martin Berglund

PhD Thesis, May 2014
Department of Computing Science
Umeå University
Sweden

Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden

mbe@cs.umu.se

ISBN 978-91-7601-047-1
ISSN 0348-0542
UMINF 14.13

Cover photo by Tc Morgan (used under Creative Commons license BY-NC-SA 2.0).
Printed by Print & Media, Umeå University, 2014.

Abstract

The work presented in this thesis discusses various formal language formalisms that extend classical formalisms like regular expressions and context-free grammars with additional abilities, most relating to order. This is done while focusing on the impact these extensions have on the efficiency of parsing the languages generated. That is, rather than taking a step up on the Chomsky hierarchy to the context-sensitive languages, which makes parsing very difficult, a smaller step is taken, adding some mechanisms which permit interesting spatial (in)dependencies to be modeled.

The most immediate example is shuffle formalisms, where existing language formalisms are extended by introducing operators which generate arbitrary interleavings of argument languages. For example, introducing a shuffle operator to the regular expressions does not make it possible to recognize context-free languages like $a^n b^n$, but it does capture some non-context-free languages like the language of all strings containing the same number of as, bs and cs. The impact these additions have on parsing has many facets. Other than shuffle operators we also consider formalisms enforcing repeating substrings, formalisms moving substrings around, and formalisms that restrict which substrings may be concatenated. The formalisms studied here all have a number of properties in common.

1. They are closely related to existing regular and context-free formalisms. They operate in a step-wise fashion, deriving strings by sequences of rule applications of individually limited power.

2. Each step generates a constant number of symbols and does not modify parts that have already been generated. That is, strings are built in an additive fashion that does not explode in size (in contrast to e.g. Lindenmayer systems). All languages here will have a semi-linear Parikh image.

3. They feature some interesting characteristic involving order or other spatial constraints. In the example of the shuffle multiple derivations are in a sense interspersed in a way that each is unaware of.

4. All of the formalisms are intended to be limited enough to make an efficient parsing algorithm at least for some cases a reasonable goal.

This thesis will give intuitive explanations of a number of formalisms fulfilling these requirements, and will sketch some results relating to the parsing problem for them. This should all be viewed as preparation for the more complete results and explanations featured in the papers given in the appendices.

Sammanfattning

Denna avhandling diskuterar utökningar av klassiska formalismer inom formella språk, till exempel reguljära uttryck och kontextfria grammatiker. Utökningarna handlar på ett eller annat sätt om ordning, och ett särskilt fokus ligger på att göra utökningarna på ett sätt som dels har intressanta spatiala/ordningsrelaterade effekter och som dels bevarar den effektiva parsningen som är möjlig för de ursprungliga klassiska formalismerna. Detta står i kontrast till att ta det större steget upp i Chomsky-hierarkin till de kontextkänsliga språken, vilket medför ett svårt parsningsproblem.

Ett omedelbart exempel på en sådan utökning är s.k. *shuffle*-formalismer. Dessa utökar existerande formalismer genom att introducera operatorer som godtyckligt sammanflätar strängar från argumentspråk. Om shuffle-operator introduceras till de reguljära uttrycken ger det inte förmågan att känna igen t.ex. det kontextfria språket $a^n b^n$, men det fångar istället vissa språk som inte är kontextfria, till exempel språket som består av alla strängar som innehåller lika många a:n, b:n och c:n. Sättet på vilket dessa utökningar påverkar parsningsproblemet är mångfacetterat. Utöver dessa shuffle-operatorer tas också formalismer där delsträngar kan upprepas, formalismer där delsträngar flyttas runt, och formalismer som begränsar hur delsträngar får konkateneras upp. Formalismerna som tas upp här har dock vissa egenskaper gemensamma.

1. De är nära besläktade med de klassiska reguljära och kontextfria formalismerna. De arbetar stegvis, och konstruerar strängar genom successiva applikationer av individuellt enkla regler.

2. Varje steg genererar ett konstant antal symboler och modifierar inte det som redan genererats. Det vill säga, strängar byggs additivt och längden på dem kan inte explodera (i kontrast till t.ex. Lindenmayer-system). Alla språk som tar upp kommer att ha en semi-linjär Parikh-avbildning.

3. De har någon instressant spatial/ordningsrelaterad egenskap. Exempelvis sättet på vilket shuffle-operatorer sammanflätar annars oberoende deriveringar.

4. Alla formalismera är tänkta att vara begränsade nog att det är resonabelt att ha effektiv parsning som mål.

Denna avhandling kommer att ge intuitiva förklaring av ett antal formalismer som uppfyller ovanstående krav, och kommer att skissa en blandning av resultat relaterade till parsningsproblemet för dem. Detta bör ses som förberedande inför läsning av de mer djupgående och komplexa resultaten och förklaringarna i de artiklar som finns inkluderade som appendix.

Preface

This thesis consists of an introduction which discusses some different language formalisms in the field of formal languages, touches upon some of their properties and their relations to each other, and gives a short overview of relevant research. In the appendix the following six articles, relating to the subjects discussed in the introduction, are included.

Paper I Martin Berglund, Henrik Björklund, and Johanna Björklund. Shuffled languages – representation and recognition. *Theoretical Computer Science*, 489-490:1–20, 2013.

Paper II Martin Berglund, Henrik Björklund, and Frank Drewes. On the parameterized complexity of Linear Context-Free Rewriting Systems. In *Proceedings of the 13th Meeting on the Mathematics of Language (MoL 13)*, pages 21–29, Sofia, Bulgaria, August 2013. Association for Computational Linguistics.

Paper III Martin Berglund, Henrik Björklund, Frank Drewes, Brink van der Merwe, and Bruce Watson. Cuts in regular expressions. In Marie-Pierre Béal and Olivier Carton, editors, *Proceeding of the 17th International Conference on Developments in Language Theory (DLT 2013)*, pages 70–81, 2013.

Paper IV Martin Berglund, Frank Drewes, and Brink van der Merwe. Analyzing catatrophic backtracking behavior in practical regular expression matching. Submitted to the *14th International Conference on Automata and Formal Languages (AFL 2014)*, 2014.

Paper V Martin Berglund. Characterizing non-regularity. Technical Report UMINF 14.12, Computing Science, Umeå University, `http://www8.cs.umu.se/research/uminf/`, 2014. In collaboration with Henrik Björklund and Frank Drewes.

Paper VI Martin Berglund. Analyzing edit distance on trees: Tree swap distance is intractable. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2011*, pages 59–73. Prague Stringology Club, Czech Technical University, 2011.

Acknowledgments

I must firstly thank my primary advisor, Frank Drewes, who made all this both possible, enjoyable and inspiring. In much the same vein I thank my co-advisor, Henrik Björklund, who knows many things and throws a good dinner party, as well as my unofficial co-advisor Johanna Björklund, who organizes many things and makes people have fun when they otherwise would not. I must also thank the rest of my university colleagues, in the Natural and Formal Languages Group (thanks to Niklas, Petter and Suna) and many others in many other places. A special thank you to all the support and administrative staff at the department and university, who have helped me out with countless things on countless occasions, a fact too easily forgotten. I also owe a great debt to all my research collaborators outside of this university, including but not limited to Brink van der Merwe and Bruce Watson. I thank those who have given me useful research advice along the way, like Michael Minock and Stephen Hegner.

On the slightly less professional front I thank my family for their support, in particular in offering places and moments of calm when things were hectic. I thank my friends who have helped both distract from and inspire my work as appropriate, thanks to, among many others, Gustaf, Sandra, Josefin, Sigge, Mårten, John, a Magnus or two, some Tommy, perhaps a Johan and a Maria, and many many more.

I wish to dedicate this work to the memory of Holger Berglund and Bertil Larsson, both of my grandfathers, who passed away during my studies leading up to this thesis.

Contents

Chapter 1
Introduction

This thesis studies extensions of some classical formal languages formalisms, notably for the regular and context-free languages. The extensions center primarily around additions of operations or mechanism that constrain or loosen order, with a special focus on parsing in the presence of such ordering loosening or constraints. This statement is, of course, quite vague. The extensions take such a form that they modify the way in which a grammar or automaton generates a string. "Order" here refers to a spatial view of this generation.

Very informally, imagine a person with finite memory (a natural assumption) who is tasked to write down certain types of strings of symbols on paper. The ways in which he or she is allowed to move around the paper will impact the types of strings they can write. If they are required to start at the left (i.e., start with the first, leftmost, symbol) and work their way through the string in a left-to-right fashion they can easily write the string $abcabcabc\ldots$, but the strings $\{ab, aabb, aaabbb, \ldots\}$ (i.e. as followed by an equal number of bs) require them to remember the number of as written if it is done in a left-to-right fashion, which is arbitrarily much information to remember. If the person is permitted to keep track of the middle of the string, adding symbols on the right and left side simultaneously, they can easily write strings of the second type by simply in each step writing one a and one b, never having to remember how many steps have been made. The first variant, where the person has to work left-to-right and cannot remember arbitrarily much is an informal description of finite automata, a characterization of the very important class of regular languages. The case where the person keeps track of the middle and writes on both the left and the right corresponds to the class of linear context-free languages, another very classical concept. From this perspective it is easy to imagine additional extensions of the formalisms, a notable example is that the writer may remember *multiple* positions, and add symbols to them interchangeably, which corresponds to a more complex language class.

Among the variety of formalisms one can imagine that modify the way in which generation happens it is important to remain true to the spirit of classical mechanisms. This tends to return to the idea that only finite memory is required when viewed from the correct perspective. Consider for example the following trivial formalism.

Example 1.1 (Mappings of copy-languages) Given two mappings σ_1, σ_2 from $\{a, b\}$ to arbitrary strings and a string w decide whether there exist some $\alpha_1, \ldots, \alpha_n \in \{a, b\}$ such that $\sigma_1(\alpha_1)\cdots\sigma_1(\alpha_n) \cdot \sigma_2(\alpha_1)\cdots\sigma_2(\alpha_n) = w$. ◇

This particular example is simplified quite a bit, but there are popular formalisms exhibiting this exact behavior, where some underlying "decision" is made in one derivation step, and the result gets reflected in multiple (but normally constant number of) places in the output string. The mapping may make it difficult to actually recognize the decision after the fact, but the problem is very related to parsing for some language classes with similar spatial dependencies.

Not all formalisms are concerned with instilling this extra level of order on the string, we also consider cases where separate "underlying decisions" may become intertwined or otherwise not get spatially separated in the way we are used to. Consider the following example of a fairly important real-world problem where difficulties arise from insufficient order.

Example 1.2 (Parallel program verification) Let P be a computer program which when run produces some output string. Assume we have a context-free grammar G which is such that if a string w can be output by a correct run of P then w can be derived in G. Then, whenever P produces output that is not accepted by G we know that P is not functioning properly.

Now run n copies of the program P, in parallel, all producing output simultaneously into the same string w. In w the outputs of the different instances of P will be arbitrarily interleaved. Now we wish to use G to determine whether this w is consistent with n copies of P running correctly. ◇

The *lack* of order makes this problem difficult, to answer the question we need to somehow track how single decisions in single instances of the program may have been spread out across the resulting string. As these artifacts may be arbitrarily far apart this problem becomes rather difficult, and the unfortunate reality is that the string w may *appear* consistent despite a program failing to run in accordance with G, due to some other part of the string masking the fault.

The cases in Example 1.1 and Example 1.2 are almost each others opposites, but are connected in that they are both possible to describe by a spatial dependence in the strings. A simple block-wise dependence in Example 1.1, and an entirely scattered dependence in Example 1.2.

Earlier Work This work is deeply related to the preceding licentiate thesis [Ber12] by the same author. While this thesis is intended to replace this earlier work it may for some readers be of interest to refer back to [Ber12] for further examples and explanations of many of the same concepts.

1.1 Formal Languages

Formal languages is a vast area of study, it covers both a lot of practical algorithmic work with numerous application areas, as well as more theoretically founded mathematical study. The original subject of study in formal languages are string languages. These are concerned with sequences of symbols from a finite alphabet, which is usually denoted Σ. Going forward we will usually simply assume that Σ is the latin alpha-

bet, $\Sigma = \{a, b, c, \ldots, z\}$, meaning that usual words like "cat" and "biscuit" are strings in this formal sense. We let ε denote the empty string. A *language* is a, potentially infinite, set of strings. One trivial example is the empty set, $\varnothing$, the language that contains no strings, and the set of *all* strings, which we denote Σ^*. Other examples include finite languages like $\{cat\}$ and $\{cat, biscuit\}$, infinite languages like the set of all strings *except* "cat", the language $\{ab, aabb, aaabbb, aaaabbbb, \ldots\}$, and, over the alphabet $\{0, \ldots, 9\}$, the language $\{3, 31, 314, 3141, 31415, 314159, \ldots\}$.

The most immediate subject of study in formal languages is representing them. Finite languages like $\varnothing$ and $\{cat, biscuit\}$ are easy to describe by exhaustively enumerating the strings they contain. Some infinite languages are also trivial, the language containing all strings except "cat" can be described by enumerating the strings it does not contain. However, languages like $\{ab, aabb, aaabbb, aaaabbbb, \ldots\}$ and $\{3, 31, 314, 3141, 31415, \ldots\}$ are more complex. Certainly the "dots"-notation used here to describe them is flawed, as the generalization intended is ambiguous at best.

This question of representation for languages is the core of formal language theory, arbitrary languages can of course represent almost arbitrary computational problems, but the question of how the language can be *finitely represented* restricts matters. Specifically what is studied is *classes* of languages defined by the type of descriptional mechanism capable of capturing them. Most trivially, the finite languages is a language class, defined by being describable through simply enumerating the strings.

While language classes are typically defined using the formalism that can describe them it is important to remember that languages are abstract entities that exist in and of themselves. In most formalisms a given language can be represented by many different grammars or automata, and few of the usual formalisms have unique normal forms that can be computed.

1.2 An Example Representation

To make the previous more concrete let us establish a representation for formal language formalisms as rather visual grammars. We call these instances of formalisms "grammars" here, but the sketches used here intentionally straddle the boundary of what is traditionally called "grammars" and what is called "automata".

1.2.1 Our Grammar Sketch

Essentially the grammars will consist of two parts; "memory", or state, and rules. *States*, or *non-terminals*, represent what the formalism is remembering about the string it is generating. They are simply symbols attached to the intermediary output. The grammars always start out in the state *S*, the *initial non-terminal* in an otherwise empty string. The rules specify which state can generate what in the string. We write the rules down as shown in Figure 1.3, where three rules are given which generate the language $\{a, aba, ababa, abababa, \ldots\}$ using two non-terminals. The left-hand side shows the state which the rule applies to. The little dot below the *S* represents the position in the string the *S* is keeping track of. On the right-hand side is shown what the formalism generates, in the case of the first rule it outputs the symbol "a", followed by a position

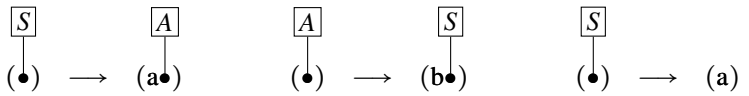
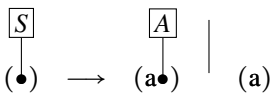


Figure 1.3: A regular grammar generating the language $\{a, aba, ababa, abababa, \ldots\}$ using three rules. S is the initial non-terminal.

which is kept track of by the second non-terminal A. In effect S "remembers" that the next symbol should be an "a", and the second non-terminal A remembers that the next symbol should be a "b" (and we then go back to S. The third rule allows the S to generate a final "a" and ending the generation by producing no new non-terminal. Since the first and third rule have the same left-hand side the abbreviation



is sometimes used in place of writing both out in full. We write the generation of strings in the way shown in Figure 1.4, where a derivation is performed using the grammar from Figure 1.3 to generate the string "ababa". Notice that, as usual, none

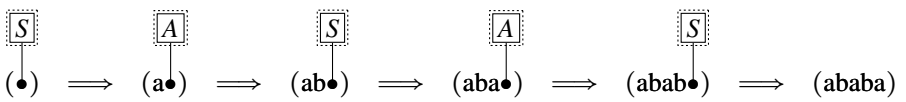


Figure 1.4: A derivation of the string "ababa" using the grammar Figure 1.3. The derivation starts with the initial non-terminal S, applies the first rule, this produces the non-terminal A, making the second rule the only possible one. This is then repeated, and finally the third rule is used to get rid of the non-terminal S entirely. As there is no more state left the derivation is finished, and the string "ababa" has been generated. The dotted outline around non-terminals show which non-terminal is used in the next rule application, but as there is only one to choose from in each step it is not very informative here.

of the intermediary strings are "generated", all states must be gone before generation is finished. The black bullets, or "positions" act as the points of the string tracked by attached non-terminals. Their role will become slightly more complex later on.

1.2.2 Generating Regular Languages

A simple and important class of languages that we can generate with grammars of the type we have sketched are the regular languages. Specifically the regular languages are precisely the following.

Definition 1.5 (Regular Grammars) A grammar of the form sketched in Figure 1.3 is regular if

- It is finite.
- Each right-hand side contains zero or one symbol from Σ and zero or one non-terminal attached to the position (bullet).
- The position is to the right of the symbol if one exists.

Every regular language can be represented by a grammar of this form. ◇

A grammar G then generates exactly the strings one can produce by starting from S attached to the initial position, and then repeatedly picking a rule, and replacing an instance of the non-terminal on the left-hand side of the rule (this is then only possible if that non-terminal exists in the string) by the new substring on the right-hand side of the rule. If a point is reached where no non-terminal exists in the string the generated string w is in the language, denoted $w \in \mathcal{L}(G)$. That is, $\mathcal{L}(G)$ is a set consisting of exactly these strings.

1.2.3 Regular Expressions as an Alternative

A regular expression is another way of expressing a language, which is equivalent to the description of a regular grammar in Definition 1.5, but which is often more compact and convenient, as well as being very popular in practical use.

Definition 1.6 (Regular Expressions) A regular expression over the alphabet Σ is, inductively, the following. For each $\alpha \in \Sigma$ and regular expressions R and T:

- ε is a regular expression with $\mathcal{L}(\varepsilon) = \{\varepsilon\}$.
- α is a regular expression with $\mathcal{L}(\alpha) = \{\alpha\}$.
- $R \cdot T$ is a regular expression with $\mathcal{L}(R \cdot T) = \{wv \mid w \in \mathcal{L}(R), v \in \mathcal{L}(T)\}$ (i.e. the concatenation of the strings in the languages of the subexpressions). We often write RT as an abbreviation.
- $R|T$ is a regular expression, with $\mathcal{L}(R|T) = \mathcal{L}(R) \cup \mathcal{L}(T)$.
- R^* is a regular expression, with $\mathcal{L}(R^*) = \{\varepsilon\} \cup \{wv \mid w \in \mathcal{L}(R), v \in \mathcal{L}(R^*)\}$ inductively. That is, the concatenation of arbitrarily many strings from R. ◇

1.3 Computational Problems in Formal Languages

With formalisms for representing formal languages in hand it is time to consider the various questions that can be asked about them. An immediate example is the emptiness problem; given a grammar G, does it generate the language $\varnothing$? Computing the answer to this problem is easy for context-free languages[1], but it is undecidable to determine if a context-free language generates Σ^*, the language of *all* strings.

[1] We have not defined the context-free languages properly, but all regular languages are context-free, and some context-free languages are not regular, so it can serve as an unspecific more powerful example.

Many problems also deal with languages themselves, being somewhat independent of representation. For example, given two context-free languages (i.e., two languages that can be generated by some context-free grammar) L and L', is the language $L \cup L'$ also context-free? It, in fact, is, and given any context-free grammar for L and L' a grammar for $L \cup L'$ can easily be constructed. The same does *not* hold for the language $L \cap L'$, some context-free languages have an intersection that is not context-free. The regular languages, however, as closed under intersection, so for all regular languages L and L' the language $L \cap L'$ is regular as well, a fact we will make use of later.

It is important to remember that while grammars may determine languages the grammar is not necessarily always in the most convenient form. Given a regular grammar G it is easy to determine if it generates Σ^*, but it is hard to determine if a context-free grammar generates Σ^*. However, context-free grammars can generate all the regular languages as well, but even if a context-free grammar generates a regular language it is *still* hard to tell if it generates Σ^* (in fact, as Σ^* *is* regular this is a part of the general problem).

The problem we are primarily concerned with in this work, however, is the *membership problem*. This is the problem of determining whether a string belongs to a given language or not. There are at least three different variations of the membership problem of interest here.

Definition 1.7 (The Uniform Membership Problem) Let $\mathcal{G}$ be a class of grammars (e.g. context-free grammars) such that each $G \in \mathcal{G}$ defines a formal language. *The uniform membership problem for* $\mathcal{G}$ is "Given a string w and some $G \in \mathcal{G}$ as input, is w in the language generated by G?" ◇

This case is certainly of interest at times, but fairly often the details of the formalism $\mathcal{G}$ are irrelevant to the practical problem. The most notable example is in instances where the language is known in advance and can be coded into the most efficient representation imaginable. A second type of membership problem accounts for this case, by simply considering *only* the string part of the input.

Definition 1.8 (The Non-Uniform Membership Problem) Let L be any language. Then the *non-uniform membership problem for* L is "Given a string w as input, is w in L?" ◇

There is a third approach, called fixed-parameter analysis, which provides more nuance in the complexity analysis of the membership problems. In this approach any part of the problem may be designated the "parameter", and is considered secondary in complexity concerns. This is treated in Section 3.5.1.

The final, and perhaps most practically interesting case, is *parsing*. In parsing we no longer expect to get just a "yes" or "no" as an answer to the question whether the string belongs to the language, we expect a *description* of *why* the string belongs to the language. For example, when asking whether the string "ababa" can be generated by the grammar in Figure 1.3 the answer should not be "yes", it should be some description of the generation procedure in Figure 1.4. In most practical cases any solution to the membership problems in Definition 1.7 and 1.8 will construct some representation of this answer *anyway* (the case of Definition 1.8 becomes more complicated,

however, as the internal representation of the language may be hard to practically decipher). Thanks to this fact this thesis will primarily refer to and work on membership problems, despite it being understood that parsing is the real goal.

1.4 Outline of Introduction

In the following chapters we will look at some formalisms that are of interest for this thesis (and are studied in the papers included). We will start out using variations on the informal notation demonstrated above (as in Figure 1.3), modifying it to illustrate the general idea of how the formalisms differ. More formalized, and deeper, matters are then considered for each.

For the most part each chapter starts out with a self-contained informal introduction, with a more formal treatment being undertaken at the end. This is intended to cater to multiple types of readers. A casual reader may be most interested in reading every chapter only up until the section marked by a star, *, and then skipping to the next. The non-starred portion of the introduction is self-contained. For a deeper treatment the entirety of the introduction may be read, but, of course, in the end most of the material is in the accompanying papers, and readers familiar with the area may be best served only skimming the introduction in favor of proceeding to the papers.

Chapter 2 gives a light introduction to *shuffle* formalisms, which are related to Example 1.2, extending regular expressions with an operator that interleaves strings. This sets the scene for a short summary of the contents of Paper I, with some words on Paper V in addition. Chapter 3 discusses synchronized substrings, similar to Example 1.1, going into a summary of Paper II. Chapter 4 discusses some extensions of regular expressions, primarily dealing with the *cut* operator, which provides a more limited string concatenation, but also giving an overview of some of the details of real-world matching engines. Papers III and IV are then discussed in brief in this context. Chapter 5 discusses distance measures on languages for handling errors. This yields a short discussion of grammar-instructed block movements, where substrings may be moved around in the string depending on how they were generated by a grammar, leading into Paper VI. Finally, Chapter 6 provides a short summary.

CHAPTER 2

Shuffle-Like Behaviors in Languages

Shuffle in the title of this chapter refers to shuffling a deck of cards, specifically to the riffle shuffle, where the deck is separated into two halves, which are then interleaved. This idea, transferred to formal languages, is intended to capture situations such as the one illustrated in Example 1.2, where multiple mostly independent generations are performed in an interleaved fashion.

2.1 The Binary Shuffle Operator

We specifically transfer the riffle shuffle to the case of strings in the following way. Starting with the strings "ab" and "cd", the *shuffle* of "ab" and "cd" is denoted $ab \odot cd$, and results in the language $\{abcd, acbd, cabd, acdb, cadb, cdab\}$, that is, all ways to interleave "ab" with "cd" while not affecting the internal order of the strings. Let us make this point slightly more formal with a definition.

Definition 2.1 (Shuffle Operator) Let w and v be two arbitrary strings. Then $w \odot \varepsilon = \varepsilon \odot w = \{w\}$. Recall that ε denotes the empty string.

If both w and v are non-empty let $w = \alpha w'$ and $v = \beta v'$ (for strings w' and v', single symbols α and β). Then $w \odot v = \alpha(w' \odot v) \cup \beta(w \odot v')$. ◇

This is then generalized to the shuffle of two languages in a straightforward way, for two languages L and L' we let the shuffle $L \odot L'$ be the language of shuffles of strings in L with strings in L', or $\bigcup\{w \odot w' \mid w \in L, w' \in L\}$.

Example 2.2 (The shuffle of two languages) Let $\mathcal{L} = \{ab, abab, ababab, \ldots\}$ and $\mathcal{L}' = \{bc, bcbc, bcbcbc, \ldots\}$. Then the shuffle $\mathcal{L} \odot \mathcal{L}'$ contains, for example, *abbc* (all of "ab" which is in $\mathcal{L}$ occurring before "bc" which is in $\mathcal{L}'$), *babc* (same strings interleaved differently), and *abbabcbcabab*. ◇

2.2 Sketching Grammars Capturing Shuffle

Without further ado we can fairly easily modify the graphical grammars we previously introduced to generate shuffles of this kind. We for the moment stick to the regular

languages, such as in Figure 1.3, and then extend the formalism to combine them. There are a number of restrictions on the shape of the grammars in this formalism:

1. There may be at most one non-terminal position marker (black dot) on the right-hand side of a rule.

2. The right-hand side of a rule may contain at most one generated symbol (from Σ), and the non-terminal position marker, if there is one, must be to the right of the symbol.

These two requirements together in effect require the grammar to work from left to right, generating one symbol at a time. We now, on the other hand, permit more than one non-terminal to attach itself to the same "position" (we will also in the next section outline how a non-terminal may be attached to another). In this way (with the correct precise semantics) we arrive at shuffle formalisms of various kinds. Consider for example the grammar in Figure 2.3. Effectively this grammar will generate the

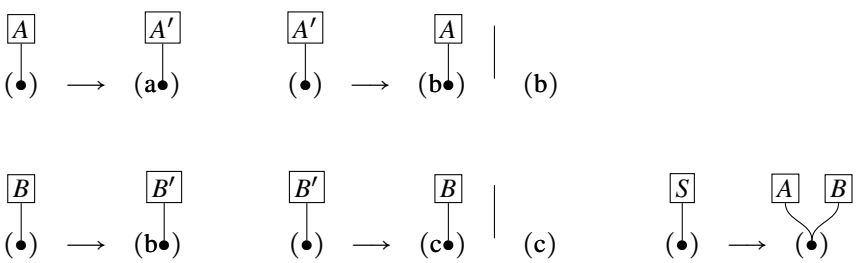


Figure 2.3: A grammar generating a language exhibiting a shuffling behavior.

shuffle $L_A \odot L_B$, if we let L_A and L_B denote the language the grammar would generate if we started with the non-terminal A and B respectively. The way the grammar works is that it starts out (since there is only one rule for the initial state) by attaching two states, A and B, to the same position. The intended semantics of this is that all non-terminals attached to the same position can generate symbols simultaneously, while the others are unaware. A derivation of the string "bacbbc" is shown in Figure 2.4.

The languages that these grammars express are closely related to the languages generated by (or, rather, denoted by) regular expressions extended with the shuffle operator. For example, the grammar in Figure 2.3 corresponds to the expression $(ab)^* \odot (bc)^*$. These expressions form a part of what is known as "shuffle expressions". This is not all there is to the grammars or to shuffle expressions. Consider the grammar in Figure 2.5. This grammar is able to keep attaching arbitrarily many additional instances of the non-terminal S to the initial position, each S can produce one "a" to transition into the non-terminal B, which simply produces a "b" and disappears. An example derivation is shown in Figure 2.6. The language generated by this grammar is, obviously, $ab \odot ab \odot ab \odot \cdots$ (the language which is such that in every prefix the number of "a"s is greater or equal to the number of "b"s, and the entire string has the same number of "a"s and "b"s). This language is not expressed by any regular

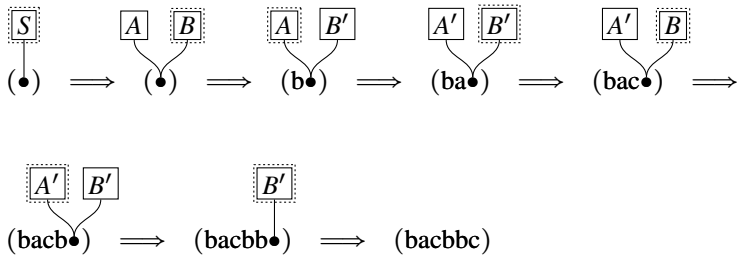


Figure 2.4: A derivation of the string "bacbbc" in the grammar from Figure 2.3. Notice that there are multiple ways this string could be derived, here the last "b" "belongs" to the string "ab" generated by the A non-terminal, but the second to last could be used instead.

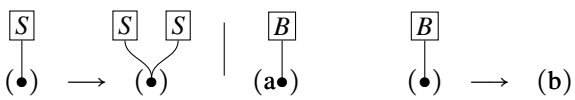


Figure 2.5: A grammar that showcases the ability to shuffle arbitrarily many strings.

expression extended by the shuffle operator, but general shuffle expressions have an additional operator for this purpose.

2.3 The Shuffle Closure

To complete the picture, shuffle expressions are regular expressions (regular expressions are introduced in short in Definition 1.6, for a more complete introduction see e.g. [HMU03]) extended with the binary shuffle operator from Definition 2.1 and the unary shuffle closure operator, denoted $\mathcal{L}^{\odot}$ (for some expression or language $\mathcal{L}$). The shuffle closure captures exactly languages of the type illustrated in Figure 2.5, where

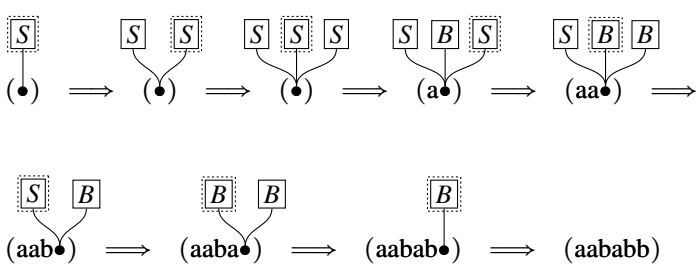


Figure 2.6: An example derivation using the grammar from Figure 2.5.

arbitrarily many strings from a language are shuffled together. Recall that $\mathcal{L}(E)$ denotes the language generated/denoted by a grammar/expression E.

Definition 2.7 (Shuffle Closure) For a language $\mathcal{L}$ the shuffle closure of $\mathcal{L}$, denoted $\mathcal{L}^{\odot}$ is $\{\varepsilon\} \cup \{w \odot \mathcal{L}^{\odot} \mid w \in \mathcal{L}\}$. For an expression E of course $\mathcal{L}(E^{\odot}) = \mathcal{L}(E)^{\odot}$. ◇

The language generated by the grammar in Figure 2.5 is then simply $(ab)^{\odot}$.

The grammatical formalism we have so far sketched can represent simple shuffles, but it is not yet complete. The shuffle expression $(ab)^{\odot}c$ causes trouble. If we start out with the grammar in Figure 2.5 (and we more or less have to) we somehow have to designate a non-terminal to generate the final c, but we have no way of ensuring that all the *other* non-terminals finish generating first. As such further extensions to the grammars are required. To leap straight to the illustrative example, see Figure 2.8. Here the first rule generates two non-terminals, one A and one C, where the C is

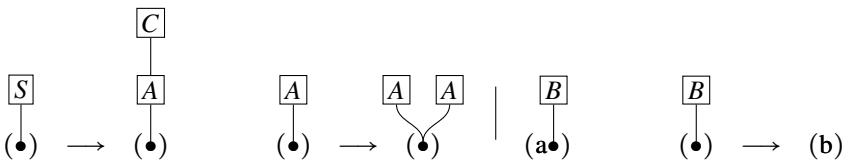


Figure 2.8: This grammar illustrates an extension which enables the combination of shuffling with sequential behavior. Specifically this grammar generates the language $(ab)^{\odot}c$.

no longer connected to the position tracked, but is rather connected to the A. We say that C *depends* on A. The semantics is that rules may only be applied to non-terminals attached only to the position, all non-terminals that depend on another must be left alone. If new non-terminals are created from the one on which C depends then C will depend on all the new non-terminals. If all non-terminals on which C depends are removed (i.e. they finish generating) then C gets attached to the position. See the example run in Figure 2.9. Notice how the C is generated with the first rule application, but then no rule can be applied to it until all the non-terminals it depends on have disappeared, meaning, in this case, that it will generate the last symbol in the string, since all the As (and subsequent Bs) much first finish.

2.4 Shuffle Operators and the Regular Languages

It may be interesting to note that a shuffle expression which uses *only* the binary shuffle operator, $\odot$, still denotes a regular language (i.e. any regular formalism, such as finite automata or regular expressions, can represent the same language). That is, we do not *need* to generate multiple non-terminals to construct a shuffle language of this kind. This is fairly easy to see, recall the simple shuffle grammar in Figure 2.3, and then consider a new grammar with non-terminals containing multiple symbols. Consider specifically the two left-most rules in that figure, and then consider the new rules in

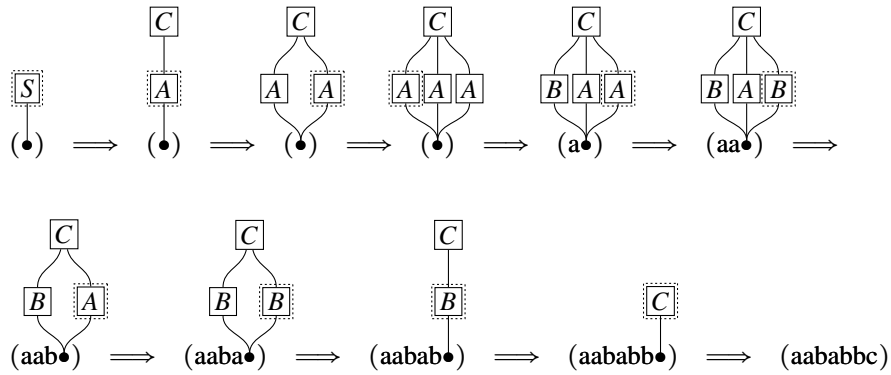


Figure 2.9: Generation of the string "aababbc" using the grammar from Figure 2.8.

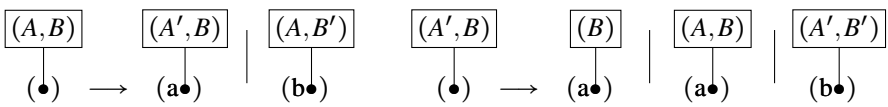


Figure 2.10: Some example rules from a regular grammar for the shuffle grammar in Figure 2.3.

Figure 2.10. That is, we create non-terminals which contain all the non-terminals of a certain step of the generation for the original grammar. The first left-hand side, with the nonterminal (A,B), corresponds to the situation created immediately after the first rule applied in Figure 2.4, and the two possible right-hand sides correspond to either applying a rule to the A or to the B. Similarly the second left-hand side corresponds to when A' and B are tracking the position, and either A' is chosen to disappear generating a, or just produce a and generate a new A, or B generates a b turning into B'. Instead of the grammar in Figure 2.3 we get a grammar with the non-terminals (S), (A,B), (A',B), (A,B'), (A',B'), (A), (A'), (B), (B'), quite a number, but this grammar only has a single non-terminal tracking the point at any point of a generation. This procedure demonstrates that only one non-terminal is necessary, so the language generated is regular. However, a potentially exponential number of non-terminals may be generated performing the construction, so this *cannot* be combined with the efficient parsing for regular languages to produce an efficient uniform parsing algorithm. This construction works for any expression with arbitrarily many binary shuffle operators, as they still only give rise to a constant number of possible sets of non-terminals attached to the tracked position, making this product construction generate a finite regular grammar.

Applying the shuffle closure, however, does not necessarily preserve regularity. Recall that the language $\{a^n b^n \mid n \in \mathbb{N}\}$ is not regular, as reading it from left to right arbitrarily much information (the number of as) must be remembered. Regular lan-

guages are also closed under intersection, so if R_1 and R_2 are regular then so is $R_1 \cap R_2$. Consider the language $\mathcal{L}(a^*b^*)$, which contains all strings consisting of some number of as followed by some number of bs. This is clearly regular. However,

$$\mathcal{L}((ab)^{\odot}) \cap \mathcal{L}(a^*b^*) = \{a^n b^n \mid n \in \mathbb{N}\}$$

since the language $\mathcal{L}((ab)^{\odot})$ only matches strings with equally many as and bs. As such, since $\{a^n b^n \mid n \in \mathbb{N}\}$ is not regular it follows that $(ab)^{\odot}$ cannot be regular either. Notice that in terms of the sketched grammars above this corresponds to the case where arbitrarily many non-terminals may be attached to the tracked position, which would create an infinite grammar if the product construction above was attempted.

2.5 Shuffle Expressions and Concurrent Finite State Automata

The formalism that these sketched grammars are trying to imitate is Concurrent Finite State Automata, one of the main subjects of Paper I. These can represent all the languages that can be represented by shuffle expressions, in the way the previous sections sketched. They can, however, represent even more languages using one special trick: as was shown in the grammar in Figure 2.8 they are able to build "stacks" of non-terminals, where only the bottom one can be used to apply rules. By building these stacks arbitrarily high, by having rules that add more and more non-terminal on top, they are able to represent arbitrarily amounts of state (i.e. arbitrarily much information). In this way they are able to represent context-free languages, as well as the shuffle of context-free languages.

However, when this particular trick is removed we reach one of the important milestones. Understanding that the formalism is vaguely sketched so far (next chapter formalizes things further), let us nevertheless call it CFSA and make the following statement.

Theorem 2.11 (Fragment of Theorem 2 in Paper I) A language $\mathcal{L}$ is accepted by some shuffle expression if and only if it is accepted by some CFSA for which there exists a constant k such that no derivation in the CFSA has a stack of non-terminals higher than k. ◇

As such, CFSA capture both the well-known class of shuffle languages (the languages recognized by shuffle expressions), and permit additional language classes based on (possibly fragments of) context-free languages. This opens up questions about membership problems.

2.6 Overview of Relevant Literature

These types of languages featuring shuffle, and many questions relating to them, have been studied in depth and over quite some time. Arguably they started with a definition by S. Ginsburg and E. Spanier in 1965 [GS65]. The shuffle expressions, and the shuffle languages they generate have been the primary focus of this section so far. This is the

name given to regular expressions extended with the binary shuffle operator and unary shuffle closure, a formalism introduced by Gischer [Gis81]. These were in turn based on an 1978 article by Shaw [Sha78] on flow expressions, which were used to model concurrency. The proof that the membership problem for shuffle expressions is NP-complete in general is due to [Bar85, MS94], whereas the proof that the non-uniform case is decidable in polynomial time is due to [JS01].

Shuffle expressions are nowhere near the end of interesting aspects of the shuffle however, even if we restrict ourselves to the focus on membership problems. A very notable example is Warmuth and Hausslers 1984 paper [WH84]. This paper for example demonstrates that the uniform membership problem for the iterated shuffle of a single string is NP-complete. That is, given two strings, w and v, decide whether or not $w \in v \odot v \odot \cdots \odot v$. A precursor to one of the results in Paper I is due to Ogden, Riddle and Rounds, who in a 1978 paper [ORR78] showed that the non-uniform membership problem for the shuffle of two deterministic context-free languages is NP-complete (extended to linear deterministic context-free languages in Paper I).

Some additional examples of interesting literature on shuffle includes a deep study on what is known as shuffle on trajectories [MRS98], where the way the shuffle may happen is in itself controlled by a language, and axiomatization of shuffle [EB98]. For a longer list of references, see the introduction of Paper I.

2.7 CFSA and Context-Free Languages

As noted in Section 2.5 part of the purpose of concurrent finite-state automata is that they permit the modeling of context-free languages, for example the language $\{a^n b^n \mid n \in \mathbb{N}\}$ (i.e. the language where some number of as are followed by the same number of bs), something that is not captured by shuffle expressions. A grammar for this language is shown in Figure 2.12. A derivation in this grammar will simply gen-

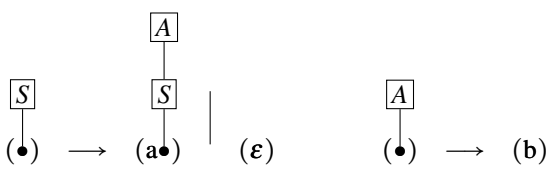


Figure 2.12: A grammar in the CFSA style for the language $\{a^n b^n \mid n \in \mathbb{N}\}$.

erate some number of as while stacking up equally many A non-terminals, then when the S is finally replaced by ε the A non-terminals drop down and each successively generates a b. In this way the (non-shuffle) language is generated. Effectively the CFSA simulates a push-down automaton.

We can easily shuffle two context-free languages in this way, by simply taking grammars of the style of Figure 2.12 and generating their initial non-terminal (now suitably renamed) attached to the same position using a new initial non-terminal rule. This type of language, mixing context-free languages and shuffle, are of some practi-

cal interest, so Paper I studies this type of situation in some depth.

In fact, where shuffle expressions are regular expressions with the two shuffle operators added, it is instructive to view general CFSA as context-free languages with the addition of the binary shuffle operator. This part requires knowledge of context-free grammars, see e.g. [HMU03]. Consider the right-most rule in Figure 2.13, which showcases all the features of CFSA. Then consider the context-free grammar which

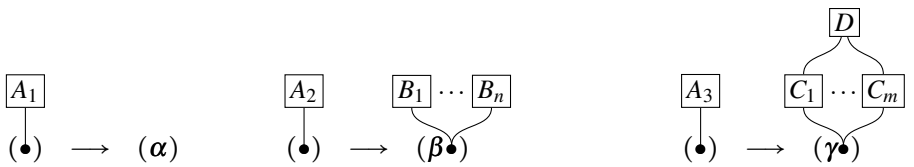


Figure 2.13: The three possible types of rules in our sketched variation of CFSA where $\alpha, \beta, \gamma \in \Sigma \cup \{\varepsilon\}$. The right-most exhibits all features, where the two first are only differentiated in that some parts don't exist.

produces strings over the alphabet $\Sigma \cup \{\odot,), (\}$ by rewriting the CFSA rules in the way shown in Table 2.14. Constructing a context-free grammar in this way, starting from a

Table 2.14: Context-free rules for the CFSA rule in Figure 2.13.

First rule	$A_1 \to \alpha$
Second rule	$A_2 \to \beta(B_1 \odot \cdots \odot B_n)$
Third rule	$A_3 \to \gamma(C_1 \odot \cdots \odot C_m)D$

CFSA A, one gets a context-free language L containing shuffle expressions which are such that $\mathcal{L}(A) = \bigcup\{\mathcal{L}(e) \mid e \in L\}$. That is, when the result of evaluating all the shuffle expressions in L are unioned together we arrive at the language generated by A.

This should serve to illustrate that all languages generated by CFSA can be viewed as "disordered" context-free languages. The above procedure generates a characterizing context-free language, which specifies which strings are to be shuffled together to produce strings in the original CFSA. As such, for example the language $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ cannot be generated by a CFSA, as it is not context-free, nor can one arrive at it by relaxing the order of substrings in a context-free language.

2.8 Membership Problems

The membership problem for these shuffle formalisms should be divided into two parts; the membership problem for shuffle expressions, which do not feature the context-free abilities of full CFSA, and the one for full CFSA.

2.8.1 The Membership Problems for Shuffle Expressions

The membership problem for shuffle expressions is already a fairly complex question. There is a sizable body of literature, and Paper I studies one fragment of the problem.

- The non-uniform membership problem is decidable in polynomial time [JS01]. The algorithm relies on permitting each symbol read (or generated) to produce some large number of potential states, which limits the complexity in terms of the length of the string but explodes the complexity in terms of the size of the expression.
- Unsurprisingly, in view of the above, the general uniform membership problem is NP-complete [Bar85, MS94].

These two pieces paint a fairly clear picture; if we wish to check membership (or parse) a string with respect to a shuffle expression it can be done reasonably efficiently if the string is much larger than the shuffle expression. However, this does not reveal the exact way in which the complexity depends on the expression. Notably, regular expressions are (trivially) shuffle expressions, and for regular expressions the uniform membership problem is not very difficult. Paper I explores how the structure of the expression affects the complexity of the problem. See Section 2.9.

2.8.2 The Membership Problems for General CFSA

The membership problem for CFSA is NP-hard even in very restrictive cases, such as where at most two non-terminals are ever attached to a position. It may therefore be surprising that the problem is *in* NP. The overall construction hinges on limiting the size of the trees of non-terminals generated by parsing a certain string, which relies on a careful case-by-case analysis of symmetries in how non-terminals may be generated. This means that even if far more (seemingly) complex CFSA are considered the problem does not become substantially harder. All of this is treated in Paper I, which Section 2.9 now takes a deeper look into.

2.9 Contributions In the Area of Shuffle☆

This section provides, as denoted by the star, a slightly more formal treatment of the contributions to the area of shuffle that have been made in (the papers included in) this work. We need some additional definitions to start with.

2.9.1 Definitions and Notation

Let $\mathbb{N}_+$ denote $\mathbb{N} \setminus \{0\}$. A *tree* with labels from an alphabet Σ is a function $t{:}N \to \Sigma$, where $N \subseteq \mathbb{N}_+^*$ is a set of nodes which are such that

- N is prefix-closed, i.e., for every $v \in N$ and $i \in \mathbb{N}_+$, $vi \in N$ implies that $v \in N$, and
- N is closed under less-than, i.e., for all $v \in \mathbb{N}_+^*$ and $i \in \mathbb{N}_+$, $v(i+1) \in N$ implies $vi \in N$.

Let $N(t)$ denote the set of nodes in the tree t. The root of the tree is the node ε, and vi is the ith child of the node v. t/v denotes the tree with $N(t/v) = \{w \in \mathbb{N}_+^* \mid vw \in N(t)\}$ and $(t/v)(w) = t(vw)$ for all $w \in N(t/v)$. The empty tree, denoted t_ε, is a special case, since $N(t_\varepsilon) = \varnothing$ it cannot be a subtree of another tree. Given trees $t_1, \ldots, t_n$ and a symbol α, we let $\alpha[t_1, \ldots, t_n]$ denote the tree t with $t(\varepsilon) = \alpha$ and $t/i = t_i$ for all $i \in \{1, \ldots, n\}$. The tree $\alpha[\,]$ may be abbreviated by α. Given an alphabet Σ, the set of all trees of the form $t : N \to \Sigma$ is denoted by T_Σ. For trees t, t' and $v \in N(t)$ let $t_{v \mapsto t'}$ be the tree resulting from replacing the node at v by t' in t. That is, $t_{\varepsilon \mapsto t'} = t'$, and $t_{iv \mapsto t'} = t(\varepsilon)[t/1, \ldots, (t/i-1), (t/i)_{v \mapsto t'}, (t/i+1), \ldots, t/n]$ for $iv \in N(t)$ and $i \in \mathbb{N}_+$. For $t_{v \mapsto t_\varepsilon}$ the subtree at v is deleted (e.g. $\alpha[t_1, t_2, t_3]_{2 \mapsto t_\varepsilon} = \alpha[t_1, t_3]$).

2.9.2 Concurrent Finite State Automata

With this we can make a formal definition of the concurrent finite state automata already sketched. These automata are the subject at the heart of Paper I.

Definition 2.15 A *concurrent finite state automaton* is a tuple $A = (Q, \Sigma, S, \delta)$ where Q is a finite set of states, Σ is the input alphabet, $S \in Q$ is the initial state, and $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times T_Q$ are the rules.

A derivation in A is a sequence $t_1, \ldots, t_n \in T_Q$ such that $t_1 = S[\,]$ and $t_n = t_\varepsilon$. For each $i < n$ the step from $t = t_i$ to $t' = t_{i+1}$ is such that there is some $(q, \alpha, t'') \in \delta$ and $v \in N(t_i)$ such that $t/v = q[\,]$ and $t' = t_{v \mapsto t''}$. Applying this rule reads the symbol α (nothing if $\alpha = \varepsilon$). $\mathcal{L}(A)$ is the set of all strings that can be read this way.

We only permit four types of rules in δ. Deleting rules of the form $(q, \varepsilon, t_\varepsilon) \in \delta$. Horizontal rules of the form $(q, \alpha, q'[\,]) \in \delta$. Vertical rules of the forms $(q, \alpha, q'[p_1]) \in \delta$ and $(q, \alpha, q'[p_1, p_2]) \in \delta$. Finally the closure rules, where $(q, \alpha, q'[p_1, \ldots, p_1]) \in \delta$ for every number of repetitions of p_1s, greater or equal to zero. ◇

We treat the in practice infinite set of rules for the closure rules as a schema (i.e. they count as a constant number of rules for the purposes of defining the size of the automaton).

Using this definition it should be easy to see how the rules in Figure 2.13 can be constructed. The graphical rules cheat by ignoring the possibility that $\alpha = \varepsilon$, while permitting e.g. generating siblings without a root (effectively having rules $(q, \alpha, p_1 p_2)$), but it is trivial to add an additional state that serves as root for the subtree with only a deleting rule defined.

Notice that the rules overlap a bit, in that the closure schema is unnecessary if we are allowed to replace $(q, \alpha, q'[p_1, \ldots, p_1])$ with $(q, \alpha, q'[q'', p_1])$ where q'' is a new state with only two rules, $(q'', \varepsilon, t_\varepsilon)$ and $(q'', \varepsilon, q''[q'', p_1])$. However, the context-free languages are precisely those that can be recognized by a CFSA where every $(q, \alpha, t) \in \delta$ has no node with more than one child in t, and we often wish to syntactically restrict CFSA to not permit context-free languages, recreating the shuffle languages. We do this as follows: a configuration is acyclic if for every $v \in N(t)$ it holds that $t(v)$ does not occur in t/vi for any i, the shuffle languages are then precisely the CFSA where all configurations are acyclic. The closure-free shuffle languages are those recognizable by a CFSA with a finite (schema-free) δ and all reachable configurations acyclic.

2.9.3 Properties of CFSA

Paper I proves a number of relevant properties about CFSA. Notably they are closed under union, concatenation, Kleene closure, shuffle, and shuffle closure (i.e., if A and A' are CFSA then there exists a CFSA A'' such that e.g. $\mathcal{L}(A'') = \mathcal{L}(A) \odot \mathcal{L}(A')$), but not under complementation or intersection (so there exists some A and A' such that there exists no CFSA recognizing the language e.g. $\mathcal{L}(A) \cap \mathcal{L}(A')$). Emptiness of CFSA is decidable in polynomial time, and the CFSA generate only context-sensitive languages.

2.9.4 Membership Testing CFSA

Membership in general CFSA. With this done we can consider uniform membership testing for general CFSA, one of the core results of Paper I. Since even a severely restricted case of CFSA already have a NP-complete uniform membership problem [Bar85, MS94], which serves as a lower bound, it is a pleasant surprise that the general problem is *in* NP, as the restricted cases appear so relatively restrictive. A non-deterministic polynomial time algorithm can simply guess which rules to apply to accept a string, as long as the number of rules necessary (i.e. the sequence $t_1, \ldots, t_n$ in Definition 2.15) is polynomial in the length of the string. The only way this might *not* happen is if a lot of ε-rules are required. A simple polynomial rewriting procedure on A solves this, based on statements such as "if rules from δ can rewrite $q[\,]$ into $q'[\,]$ without reading a symbol, include $(q, \varepsilon, q'[\,])$ in δ." This ensures that if a derivation of a string exists in A then a *short* one exists.

Membership in the shuffle of shuffle languages and context-free languages. The CFSA model goes on to be used to prove a number of other membership problem results. One interesting case is the shuffle of a shuffle language and a context-free language, i.e., membership for the CFSA where every configuration tree (except the first one and the last one where things are getting set up and dismantled) is of the form $q[t_1, t_2]$ where t_1 is acyclic and $N(t_2) \subset 1^*$ (that is, no node in t_2 has more than one child). This proof is rather more involved, and relies on finding a number of symmetries in the way the tree corresponding to the shuffle language (i.e. t_1 here) can behave. Notably it relies on defining an equivalence relation on nodes in the tree, i.e., if we have $t(v) = t(v')$ what we do to v and v' is interchangeable. Most notably, if we in two places apply a rule schema $(q, \alpha, q'[p_1, \ldots, p_1])$ there is *no point* in generating p_1 instances in both places, we might as well pick one of the places and generate *all* the instances of p_1 necessary. In fact, in the procedure we can just remember "as long as this node is still here we can assume we have any necessary number of p_1 instances". In this way the number of possibilities are limited in such a way that a Cocke-Younger-Kasami-style table can be established for parsing. While polynomial the degree of the polynomial is very substantial, an efficient algorithm is left as future work.

The hardness of context-free shuffles. Another of the core results of Paper I is a proof that there exist two deterministic linear context-free (DLCF) languages L and L'

such that the membership problem for $L \odot L'$ is NP-complete. That is, the non-uniform membership problem for the shuffle of two DLCF languages is NP-complete. The proof relies on the following. We can construct a DLCF language L which consists of strings of the following form:

$$\underbrace{[0][1]\cdots[1][1]}_{C_1}\$\underbrace{[0][1]\cdots[1][1]}_{C_2}\$\cdots\$\underbrace{[0][1]\cdots[1][0]}_{C_2'}\$\underbrace{[0][1]\cdots[1][1]}_{C_1'}$$

where each bit-string is a polynomial-length Turing machine configuration, and C_1' is the (reversed) configuration the Turing machine reaches taking *one step* from C_1, and similarly C_2' is one step from C_2 (and so on nested inwards). The rules of the Turing machine are encoded in L. The language class is not powerful enough to relate C_1 and C_2, all it can do by itself is take a single step. We can however also construct a DLCF language L' which recognizes all strings

$$\$\underbrace{[0][1]\cdots[0][1]}_{P_1}\$\underbrace{[1][1]\cdots[1][1]}_{P_2}\$\cdots\$\underbrace{[1][1]\cdots[1][1]}_{P_2'}\$\underbrace{[1][0]\cdots[1][0]}_{P_1'}$$

which are such that P_1' is P_1 reversed, and P_2' is P_2 reversed, and so on inwards. At the center there is one extra string of the form $([0]|[1])^*$, entirely arbitrary. Now construct the string

$$\underbrace{[0]\cdots[0]}_{I}\$[[01]]\cdots[[01]]\$\$\cdots\$\$[[01]]\cdots[[01]]$$

where I is filled with the initial Turing machine configuration we are interested in. Then check if this string is in $L \odot L'$. What will happen is that L and L' will have to "share" every $[[01]]\cdots[[01]]$ substring (since neither can by itself produce e.g. $[[$), each producing half the brackets and binary digits, forcing the other to produce its *complement*. The initial I must be produced by L, as L' requires a leading \$, which makes L produce the result of taking the first step of the Turing machine in the last $[[01]]\cdots[[01]]$ section, which leaves the complement for L' to produce in the last section, which will make it produce the complement in the first $[[01]]\cdots[[01]]$ section, forcing L to produce the *same* configuration in that first sectiont that it produced in the last section. This makes it produce the result of taking another computation step in the second-to-last $[[01]]\cdots[[01]]$ section, which L' then copies, and so on. In this way the shuffle will cooperate to perform an arbitrary (non-deterministic) Turing machine computation for polynomially many steps, making the membership problem NP-hard. This is non-uniform as the Turing machine coded in L may be one of the universal machines, which reads its program from the input I.

2.9.5 The rest of Paper I.

Paper I has a number of further results, including a fixed parameter analysis of parsing shuffle expressions with the number of shuffle operators which is discussed in brief in Section 3.5.1. In addition the paper discusses the uniform membership problem for

the shuffle of a context-free language and a regular language. That is, a context-free grammar G, a finite automaton A and a string w are given as input, and the decision problem is checking whether $w \in \mathcal{L}(G) \odot \mathcal{L}(A)$). An important point in this context is that $\mathcal{L}(G) \odot \mathcal{L}(A)$ is a context-free language for all G and A. This can be shown by a simple product construction. This, however, raises a question discussed in another paper.

2.9.6 Language Class Impact of Shuffle

Paper V also considers shuffle, but here the question is of a more abstract nature. The claim studied is, for two context-free languages $L \subseteq \Sigma^*$ and $L' \subseteq \Gamma^*$ (with $\Sigma \cap \Gamma = \varnothing$) is $L \odot L' \notin \mathcal{CF}$ unless either $L \in \mathcal{Reg}$ or $L' \in \mathcal{Reg}$? That is, if the shuffle of two context-free languages is context-free must one of the languages be regular? The author conjectures that this is indeed the case, but Paper V gives only a conditional and partial proof.

CHAPTER 3

Synchronized Substrings in Languages

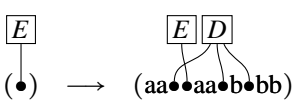
In this chapter we take a look at what can be described as formalisms with synchronized substrings. A single sequence of derivation decisions which (may) have effects in several places of a string. This is most easily illustrated by extending our running sketched formalism to generate such languages.

3.1 Sketching a Synchronized Substrings Formalism

3.1.1 The Graphical Intuition

In this section the grammars introduced in Figure 1.3 will be extended in a *different way* from the preceding shuffle chapter. In this new grammatical formalism there may *not* be more than one non-terminal attached to a position (i.e. to a bullet), *nor* may we have non-terminals depend on each other. That is, the "stacking" of non-terminals of Figure 2.8 is no longer permitted.

The new grammatical formalism for this chapter instead generalize the regular grammars in some new ways, which will pave the way to rules of the following form.



- Positions (i.e. bullets) may now occur anywhere in the string, not just at the end. There may be any number of positions on the right-hand side of rules.
- Each non-terminal may be attached to multiple positions. We say that the non-terminal tracks, or controls, those positions. This in turn means that the left-hand sides may also contain multiple positions (the number controlled by the non-terminal being replaced).

We assume that each non-terminal always tracks the same number of positions (so if A tracks 3 positions in one rule it will always track 3 positions). See Figure 3.1 for a first example of a grammar of this new kind. An example derivation using this grammar is shown in Figure 3.2.

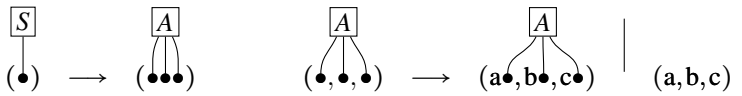


Figure 3.1: An example of a grammar of the synchronized substring variety. The initial non-terminal *S*, which tracks a single position, generates an instance of the non-terminal *A*, which tracks three positions, inserted as a string at the position which *S* was previously tracking (notice that this is *not* the same as attaching them all to that position, they are ordered and distinct in the resulting string). *A* has two rules, the first generates an *a* in the first position, a *b* in the second and a *c* in the third, while generating a new *A* tracking the positions just after each of the newly generated symbols. The last rule generates the same symbols but creates no new *A*.

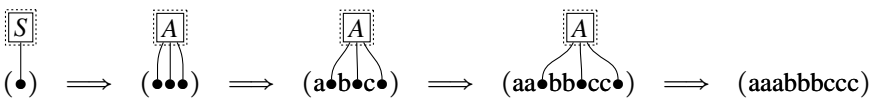
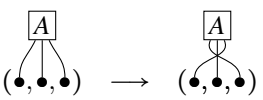


Figure 3.2: A derivation of the string "aaabbbccc" using the grammar in Figure 3.1. Notice that even though *A* tracks multiple positions there will never be commas in the derivation like there is in the grammar, the positions will instead be interspersed with real symbols in a contiguous string. Applying a rule places new substrings in some positions, and these substrings may themselves contain positions.

Notice that this formalism features ordering in the positions that the non-terminals track. Consider for example adding the following rule to the grammar in Figure 3.1.



This then permits derivations like the one shown in Figure 3.3, and more generally it permits deriving strings of the form "aacacbbbbbccaca", containing the same number of "a"s, "b"s and "c"s, but the first and last section are the same sequence with "a"s replaced with "c"s and vice versa.

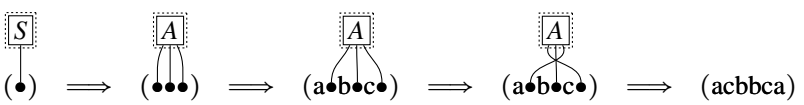


Figure 3.3: A derivation of the string "acbbca" using the grammar in Figure 3.1 extended with a rule which switches the first and third position tracked by the *A*.

3.1.2 Revisiting the Mapped Copies of Example 1.1

Example 1.1 illustrates a trivial case of synchronized substrings formalisms, where a sequence of symbols is chosen, and two different symbol mappings create two different strings, which are concatenated to produce an output string. Let us recall it here.

Example 3.4 (Mappings of copy-languages) Given two mappings σ_1, σ_2 from $\{a,b\}$ to arbitrary strings and a string w decide whether there exists some $\alpha_1, \ldots, \alpha_n \in \{a,b\}$ such that $\sigma_1(\alpha_1)\cdots\sigma_1(\alpha_n)\cdot\sigma_2(\alpha_1)\cdots\sigma_2(\alpha_n) = w$. ◇

Let us look at how

1. we can model this type of language by a grammar, and,
2. parsing may be performed, in both the uniform and non-uniform case.

3.1.3 Grammars for the Mapped Copy Languages

Here we have two alphabets, the "internal" alphabet $\Gamma = \{a,b\}$ as well as the usual Σ. In addition we have two mappings from Γ to strings in Σ. Let $w_a = \sigma_1(a)$, $w_b = \sigma_1(b)$, $v_a = \sigma_2(a)$ and $v_b = \sigma_2(b)$. Then the grammar in Figure 3.5 generates the language of the strings that the procedure in Example 1.1 yields.

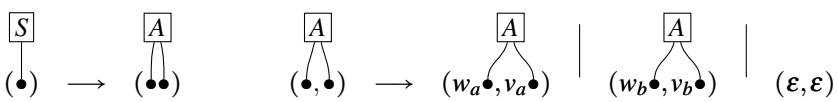


Figure 3.5: A synchronized substring-type grammar for the language that the procedure sketched in Example 1.1 can produce. Notice that w_a, v_a, w_b and v_b are strings derived from the mappings σ_1 and σ_2, rather than symbols in their own right.

3.1.4 Parsing for the Mapped Copy Languages

Let us consider the uniform parsing problem for this class of grammars (i.e., those that can be generated by some choice of σ_1 and σ_2 in the above construction). We can divide the parsing problem into two parts:

1. We need to find the position at which the concatenation happens. That is, let G be the grammar constructed as in Figure 3.5 for some σ_1 and σ_2, then, to decide if some w belongs to $\mathcal{L}(G)$ we need to tell if there is some way to divide w into two, $w = xy$, such that $\sigma_1(v) = x$ and $\sigma_2(v) = y$ for some $v \in \{a,b\}^*$.
2. The second part is finding the actual $v \in \{a,b\}^*$.

Solving the second part effectively solves the first, in the sense that if we are *given* v we will be able to tell where the concatenation happens by simply computing $\sigma_1(v)$ and $\sigma_2(v)$.

However, it might be easier to find v if the point of concatenation is found. We are in fact primarily concerned about whether parsing can be done in polynomial time or not, and if we can compute v in polynomial time given the point of concatenation we can solve the whole problem in polynomial time, as there are only linearly many positions at which the concatenation may occur. That is, we can simply for each possible way of dividing w into xy try to compute a v for this x and y. This exhaustive search at most makes the membership problem linearly more expensive.

The full algorithm for this toy example is in fact fairly simple. It will, however, serve to illustrate the more general algorithms later, where the directed graph construction will be replaced by something similar but more advanced.

Algorithm 3.6 (Parsing for Example 1.1)

```
1:  function ParseCopyMap(string w ∈ Σ*, σ1 : {a,b} → Σ*, σ2 : {a,b} → Σ*)
2:      let α1···αn = w (i.e., each αi is a symbol in Σ).
3:      let G be a digraph with nodes {(p,q) | p,q ∈ {0,...,n}} and no edges.
4:      for p,q,p′,q′ ∈ {0,...,n} do
5:          if (σ1(a) = αp+1···αp′ and σ2(a) = αq+1···αq′) or
6:             (σ1(b) = αp+1···αp′ and σ2(b) = αq+1···αq′) then
7:              add an edge from (p,q) to (p′,q′) to G
8:          end if
9:      end for
10:     for i ∈ {0,...,n} do
11:         if Reachable(G, (0,i), (i,n)) = True then
12:             return True
13:         end if
14:     end for
15:     return False
16: end function
```

Reachable is a function which takes a graph G and two nodes v and w and checks if w can be reached from v following the edges. Notice that we abuse the subscripts in the algorithm, so $\alpha_{p+1}\cdots\alpha_{p'}$ for $p \geq p'$ will be an empty string.

To quickly outline the algorithm, in lines 4–9 the graph G is constructed in such a way that a node (p',q') is only reachable from (p,q) if the substrings $\alpha_{p+1}\cdots\alpha_{p'}$ can be generated by $\sigma_1(v)$ and $\alpha_{q+1}\cdots\alpha_{q'}$ can be generated by $\sigma_2(v)$ for some common v. Once this graph is constructed lines 10–14 simply test all ways to cut the initial string into two pieces and checks on the graph if the resulting two strings can correspond to a common original string mapped through σ_1 and σ_2.

Notice that the graph will be polynomial in the size of the string to be parsed, and computing reachability on a directed graph is very cheap. Also notice that this algorithm as written is just a membership test, but making it parsing amounts to simply outputting the path in G found when line 11 succeeds.

3.2 The Broader World of Mildly Context-Sensitive Languages

The above may seem like trivialities, but it appears to be at the core of the difficulty in deciding membership for the general class of formalisms along these lines. The formalism sketched here (exemplified by the grammar in Figure 3.5) is intended to imitate a hyperedge replacement grammar (see e.g. [DHK97]) generating a string, but that formalism is equivalent to a large class of other formalisms.

3.2.1 The Mildly Context-Sensitive Category

All the formalisms discussed in this chapter fall within the category "mildly context-sensitive", defined by Aravind Joshi in [Jos85]. A language class $\mathcal{L}$ is defined by Joshi to be mildly context-sensitive if and only if all the following hold.

1. At least one language in $\mathcal{L}$ features *limited cross-serial dependencies*.
2. All languages L in $\mathcal{L}$ have a semi-linear set $\{|w| \mid w \in L\}$. That is, the lengths of strings in the language form a union of a finite number of linear sequences, $\{s_1 + ik_1 \mid i \in \mathbb{N}\} \cup \cdots \cup \{s_n + ik_n \mid i \in \mathbb{N}\}$.

In addition the following two requirements are implicit but clearly required in [Jos85]

3. All $L \in \mathcal{CF}$ are in $\mathcal{L}$, that is, a mildly context-sensitive formalism must be able to represent all context-free languages (recall Section 2.7).
4. The non-uniform membership problem for languages in $\mathcal{L}$ is decidable in polynomial time.

Requirement 1 needs some further explanation. It refers to a certain type of substring synchronization that Joshi derives from the tree-adjoining grammar formalism that he uses in that paper. The description is fairly involved, but one key detail is that languages of the form $a^n b^n c^n$ may be in such a class, but $a^n b^n a^n b^n a^n b^n \cdots$ may not. This statement may be transferred to the formalism we have sketched by noting that for every such grammar there exists some constant k such that no non-terminal tracks more than k positions, which makes it impossible to generate e.g.

$$\underbrace{a^n b^n \cdots a^n b^n}_{k+1 \text{ copies}}.$$

3.2.2 The Mildly Context-Sensitive Classes

There are at least two different classes of languages with published formalisms that fit clearly into the mildly context-sensitive definition.

1. The first is the motivating class, into which tree-adjoining grammars [JLT75] which Joshi used when first defining the category, fall. All the following formalisms are equivalent [JSW90] (that is, they define the same language class): linear indexed grammars [Gaz88], combinatorial categorial grammars [Ste87] and head grammars [Pol84].

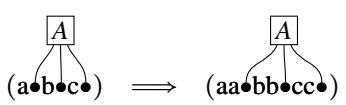
2. The second class still fulfills all the requirements outlined by Joshi, but is strictly more powerful (i.e. every language that can be generated by e.g. a head grammar can be generated by any of these formalisms). These formalisms include linear context-free rewriting systems [Wei92][1], deterministic tree-walking transducers [Wei92], multicomponent tree-adjoining grammars [Jos85, Wei88], multiple context-free grammars [SMFK91, Göt08], simple range concatenation grammars [Bou98, Bou04, BN01, VdlC02] and string-generating hyperedge replacement grammars [Hab92, DHK97].

It is interesting to note that while the mildly context-sensitive definition requires a non-uniform membership problem (i.e. the membership problem where the string, but not the grammar/automaton, is included in the input, recall Definitions 1.7 and 1.8) that is solvable in polynomial time, all the listed formalisms above have an NP-hard uniform membership problem. The way that the grammars perform concatenation, notably how many positions each non-terminal keeps track of (or, in Joshis terminology, the extent of the cross-serial dependencies), play a big part in how difficult the uniform membership problem becomes.

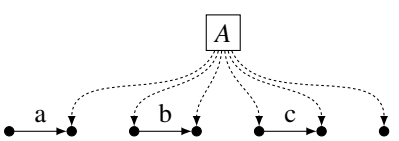
Going through all of these formalisms is not time well spent for an introductory text like this, but in the next section we will make the connection between the sketched formalism of Figure 3.1 and string-generating hyperedge replacement grammars.

3.3 String-Generating Hyperedge Replacement Grammars

The formalism sketched in Figures 3.1–3.5 is more or less a direct copy of hyperedge replacement grammars tuned for string generation. This formalism constructs a directed graph by having *hyperedges* (that is, edges that connect an arbitrary number of nodes) labeled with non-terminal symbols, and having rules that replace these by new subgraphs connected to the nodes the hyperedge was connected to. So, the rule application (using a rule from Figure 3.1)

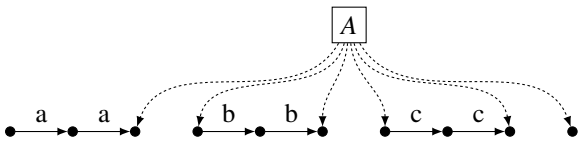


actually corresponds to rewriting the directed graph



where the box labeled by A now represents a hyperedge which is connected to 6 nodes, into this graph

[1] References are for the most part not to the original definitions, but rather to sources where they are described and related to the broader class at hand.



by the hyperedge labeled A being replaced by attaching new nodes and edges to the positions the original hyperedge was attached to, and attaching a new hyperedge (also labeled A). To make this perfectly formal we also need to number the nodes, or otherwise somehow distinguish between them, which we manage to avoid graphically in the string case by just keeping track of things left-to-right.

Notice that while the non-uniform membership problem is polynomial for *string-generating* hyperedge replacement grammars it very quickly becomes NP-hard when the grammar are allowed to generate graphs even a little bit more complex than these string-representing directed chains. In fact, if the grammar is allowed to make *multiple* chains, that is, creating a graph consisting of the union of directed chains, by simply having the hyperedge replacement rules leaving pieces unconnected, the non-uniform membership problem becomes NP-complete [LW87].

In addition note that for a hyperedge replacement grammar to generate a string it will necessarily have to keep track of both sides of each "gap" corresponding to a position, as is sketched in the above figures. If it loses track of a node that is supposed to be internal to the string it becomes impossible to later join it up to the other parts generated, and the graph becomes a set of multiple chains.

We will next take a look at a general non-uniform parsing algorithm for string-generating hyperedge replacement grammars (the informal flavor used here). This construction is from 2001 by Bertsch and Nederhof [BN01] (this is not the earliest or most efficient parsing algorithm of this type, but a straightforward and clear one).

3.4 Deciding the Membership Problem

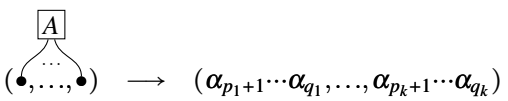
3.4.1 Deciding Non-Uniform Membership

Origins of the Algorithm The construction from [BN01] is here modified a bit for clarity and to fit into the approach we use. In its original form the algorithm takes the grammar G (for which membership should be decided) and a string w, and from these two constructs a new grammar G', which is empty if and only if $w \notin \mathcal{L}(G)$. In this way it reduces the problem of deciding membership for one grammar to the problem of deciding emptiness for another. More specifically the grammar G is one of the mildly context-sensitive formalisms (the algorithm is originally defined in terms of *Range Concatenation Grammars*, but here we opt to continue with the equivalent string-generating hyperedge replacement grammars) and the constructed grammar G' is a context-free grammar, for which emptiness testing (i.e. computing whether $\mathcal{L}(G') = \varnothing$) is very easy. However, as context-free grammars are a subset of the hyperedge replacement grammars (and emptiness testing is just as easy for hyperedge replacement grammars) we will not differentiate between the formalisms.

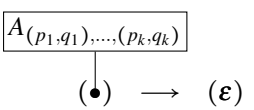
Deciding Emptiness Emptiness-checking a hyperedge replacement grammar is very easy. Start by letting all non-terminals be *unmarked*. For each rule, if the left-hand is unmarked but the right-hand side contains no unmarked non-terminals mark the non-terminal on the left-hand side and start over the looping through the rules from the first one. If we make it through all the rules without marking a non-terminal we are done, and the language generated by the grammar is empty if and only if S is still unmarked. This algorithm is clearly in $\mathcal{O}(n^2)$ where n is the number of rules, as it is a loop that is restarted at most n times (sooner or later all the non-terminals have been marked).

Reducing Membership to Emptiness It is time to solve the membership problem for string-generating hyperedge replacement grammars. Let G be the grammar, and $\alpha_1 \cdots \alpha_n$ ($\alpha_i \in \Sigma$ for each i) the string for which we wish to check whether $\alpha_1 \cdots \alpha_n \in \mathcal{L}(G)$. To decide this we will construct a new grammar G' such that $\mathcal{L}(G') \neq \emptyset$ if and only if $\alpha_1 \cdots \alpha_n \in \mathcal{L}(G)$ (specifically, $\mathcal{L}(G') = \{\varepsilon\}$ otherwise). The construction of G' should be reminiscent of the construction of the graph in Algorithm 3.6.

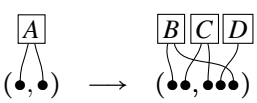
G' will be constructed in such a way that if G contains a non-terminal A which controls k positions (i.e. there are k positions on the left hand side of rules for A) then G' contains one non-terminal $A_{(p_1,q_1),\ldots,(p_k,q_k)}$ for each $p_1, q_1, \ldots\ldots, p_k, q_k \in \{0, \ldots, n\}$, such that $p_1 \leq q_1, \ldots, p_k \leq q_k$. The logic will be, if the non-terminal $A_{(p_1,q_1),\ldots,(p_k,q_k)}$ can generate *any* strings, then the non-terminal A in the original grammar G is able to generate the strings $\alpha_{p_1} \cdots \alpha_{q_1}, \ldots, \alpha_{p_k} \cdots \alpha_{q_k}$. The most direct rule to include in G' then becomes that if



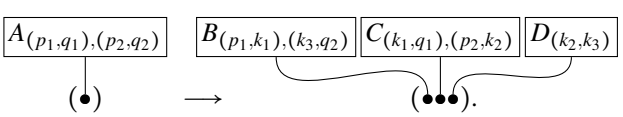
is a rule in G then



is a rule in G'. That is; if A can generate the substrings $\alpha_{p_1+1} \cdots \alpha_{q_1}$ through $\alpha_{p_k+1} \cdots \alpha_{q_k}$, then $A_{(p_1,q_1),\ldots,(p_k,q_k)}$ can generate the empty string (we could select any string as only emptiness is of interest). Similarly, for example, if there is a rule in G of the form

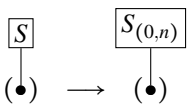


then, for all $p_1, q_1, p_2, q_2 \in \{0, n\}$, and $k_1, k_2, k_3 \in \{0, n\}$, such that $p_1 \leq k_1 \leq q_1$, and $p_2 \leq k_2, k_3 \leq q_2$ there is a rule



That is, the p_1, q_1, p_2, q_2 decide the substrings checked by the left-hand side, and k_1 is the position between p_1 and q_1 where the substring goes from being generated by the B and the C, and similarly for k_2 and k_3 with the two concatenation points in the second substring. This generalizes to arbitrary rules in the obvious way, when non-terminals and symbols are mixed one additionally needs to check that the symbols generated correspond correctly to the symbols in $\alpha_1 \cdots \alpha_n$.

As such, if each of the B, C and D non-terminals can generate their respective substrings then the right-hand side can generate the empty string, meaning that the A can generate the whole. Finally, add the rule



to G', that is, the initial non-terminal goes to the non-terminal that checks if the non-terminal S in G can generate the substring $\alpha_{0+1} \cdots \alpha_n$, i.e., the whole string.

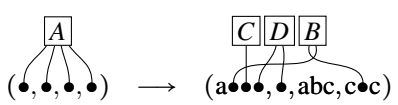
Following this procedure G' will be able to generate the empty string if and only if G can generate the string $\alpha_1 \cdots \alpha_n$. It should be clear that the size of G' is on the order of $\mathcal{O}(n^c)$ where c is polynomial in the size of the grammar G. Notably c increases with the number of positions tracked by non-terminals in G.

However, we were considering the non-uniform membership problem, and as such we view G as a constant, which in turn means that c is viewed as a constant. It follows that $\mathcal{O}(n^c)$ is a polynomial, and, as established above, deciding emptiness is polynomial, making for a membership algorithm that is polynomial.

3.4.2 Deciding Uniform Membership

Deciding the uniform membership problem for our sketched grammatical formalism appears to be extremely hard, a proof that LCFRS parsing is PSPACE-hard is given in [KNSK92]. This makes it extremely unlikely that an efficient algorithm exists. The best known algorithms (see e.g. [SMFK91] and Paper II for more references) for the problem are in $\mathcal{O}(mn^{f(r+1)})$ where m is the size of the grammar, n the size of the input string, and f and r are two very specific properties of the grammar, the *fan-out* and the *rank* of the grammar respectively. Before we get further into the explanation it is important to note that since f and r are values depending on the grammar the algorithm is in $\mathcal{O}(n^m)$ as well, that is, the length of the string raised to the size of the grammar. However, it turns out that in practice the fan-out and rank tend to be small compared to the size of the full grammar, so it is a useful distinction to separate them out.

In short, the rank of a grammar is the maximum number of non-terminals occurring on the right hand side of a rule. The fan-out is the maximum number of positions a non-terminal tracks. So for example a grammar G containing the rule



implies that it has rank at least 3, since the right-hand side has three non-terminals, and fan-out at least 4, since A controls four positions. These are "at least" since other rules may contain more non-terminals or more positions tracked.

3.4.3 On the Edge Between Non-Uniform and Uniform

So far we have seen that the problem is polynomial when the grammar is left out of the input entirely, and an algorithm that is unquestionably exponential when the grammar is included. However, this distinction, where some aspects of the grammar is separated out to illustrate that it is not exponential in the "worst" way, is a bit blunt and imprecise a way of viewing the complexity. For example, if the total number of rules were in the exponent many practical grammars would have very problematic running times, whereas if the number of symbols in the alphabet is the part in the exponent things may not be nearly as problematic.

For a deeper look in this direction, we now give a slightly deeper summary of the results in Paper II, with explanations of some of the supporting theory.

3.5 Contributions in Fixed Parameter Analysis of Mildly Context-Sensitive Languages☆

We take this opportunity to briefly explain fixed parameter complexity, as it is necessary to appreciate the contents of Paper II, and may not be common knowledge. See e.g. [FG06] for a full introduction.

3.5.1 Preliminaries in Fixed Parameter Tractability

In classical complexity theory a problem may be viewed as a set of all *positive instances* $P \subseteq I$, where I is the set of all *possible instances*. For example, we may have I be all graphs and P be the set of all graphs which have a Hamiltonian path. A decision procedure for the problem is then a program that computes a function $a : I \to \{yes, no\}$ such that $P = \{p \in I \mid a(p) = yes\}$. The running time of the program is then analyzed as a function in $|p|$. For two problems $P \subseteq I$ and $P' \subseteq I'$ a polynomial time reduction is a function $r : I \to I'$, computable in polynomial time such that $p \in P \Leftrightarrow r(p) \in P'$. A polynomial time reduction r from P to P' implies that the fastest decision procedure for P cannot be more than polynomially slower than the fastest for P'.

A parameterized problem is viewed as a set $P \subseteq I \times \mathbb{N}$, where I is again the set of all problem instances and the integer is what is called the *parameter*. A decision procedure for R again computes a function $a : I \times \mathbb{N} \to \{yes, no\}$, but now the time of deciding $a(p,k)$ is described in both $|p|$ and k. If a runs in time $f(k) \cdot |p|^{\mathcal{O}(1)}$ for *any computable function* $f : \mathbb{N} \to \mathbb{N}$ the problem is said to be fixed-parameter tractable (FPT); that is, intuitively, for a small parameter the problem is basically polynomial. The way the parameter is chosen has a large impact on how the analysis behaves. Notably, taking any problem $P \subseteq I$ and constructing the parameterized problem $\{(p, |p|) \mid p \in P\} \subseteq I \times \mathbb{N}$ yields a fixed-parameter tractable problem for every decidable P. A FPT reduction from $P \subseteq I \times \mathbb{N}$ to $P' \subseteq I' \times \mathbb{N}$ is a program which

computes a function $r : (I \times \mathbb{N}) \to (I' \times \mathbb{N})$ such that, (i) $r(p,k)$ is computable in time $f(k) \cdot |p|^{\mathcal{O}(1)}$ for some computable function f; (ii) $(p,k) \in P \Leftrightarrow r(p,k) \in P'$; and; (iii) there exists a computable function g such that for all $(p',k') = r(p,k)$ it holds that $k' \leq g(k)$.

The parameter is normally chosen as some minor aspect of the problem. A classic case is the vertex cover problem for graphs, which is NP-complete in general, but if one looks for *small* covers (i.e. does this graph of a million vertices have a cover of size five?) it turns out that the problem is easy. Vertex cover is, in fact, a classic problem in the class FPT. That is, the parameterized problem is $P \subseteq \mathbb{G} \times \mathbb{N}$, where $\mathbb{G}$ is the set of all graphs, such that $(G,k) \in P$ if and only if G has a cover of size k. This problem is decidable in time $\mathcal{O}(k|G| + 1.3^k)$, which is excellent for small k. Not all problems work out this well however, and there is a hierarchy of parameterized complexity classes:

$$\mathrm{FPT} \subseteq \mathrm{W}[1] \subseteq \mathrm{W}[2] \subseteq \cdots \subseteq \mathrm{W}[\mathrm{SAT}] \subseteq \mathrm{W}[\mathrm{P}] \subseteq \mathrm{XP}, \quad \text{where } FPT \subsetneq XP,$$

each of which has some complete (characterizing) problem.

To make a quick revisit to Paper I and Chapter 2 note that there a proof is given which shows that the uniform membership problem for shuffle expressions is $\mathrm{W}[1]$-hard when the parameter is the number of shuffle operators used in the expression. One example of an instance is $(((ab)^* \odot a^*, abb), 1)$, and deciding it involves checking whether $abb \in \mathcal{L}((ab)^* \odot a^*)$ (and checking that the expression has precisely one shuffle operator, agreeing with the parameter). It is proven that this is $\mathrm{W}[1]$-hard using a reduction from k-clique. k-clique is the set $P \subseteq \mathbb{G} \times \mathbb{N}$ where $(G,k) \in P$ if and only if G has a clique of size k. k-clique is known to be $\mathrm{W}[1]$-complete.

3.5.2 The Membership Problems of Paper II

The graphical formalism sketched in this chapter is again slightly unspecific on some details, but is close enough to hyperedge replacement string grammars that we can restate all the results in Paper II in terms of it, although care should be taken and the paper read whenever vagueness makes any statement feel unclear.

Recall the definition of *rank* and *fan-out* from Section 3.4.2. Then Paper II considers the following four parameterized membership problems, all having the set of all instances $I \times \mathbb{N}$ where $I = \{(G,w) \mid G \text{ a grammar}, w \in \Sigma^*\}$. The grammars are of the LCFRS type in Paper II, but considering the sketched hyperedge replacement grammar case is illustrative enough. In each case the decision problem is to check whether $w \in \mathcal{L}(G)$, but the parameter k differs.

1. The problem where the fan-out is constant (i.e. fixed, not part of the problem) and the rank is the parameter. That is, deciding $P \subseteq I \times \mathbb{N}$ such that $((G,w),k) \in P$ if and only if $w \in L(G)$, G has rank at most k, and G has fan-out less than or equal to a constant c.

2. The problem where the rank is constant (less than or equal to a constant c) and the fan-out is the parameter.

3. The problem where the parameter contains both the rank and the fan-out. That is, the tuples in $P \subseteq I \times \mathbb{N}$ are still $((G,w),k)$ but G has rank at most k and fan-out at most k.[2]

4. The problem where the parameter contains the rank, the fan-out, and the length of the derivation. That is, we again have $((G,w),k) \in P$ if and only if $w \in \mathcal{L}(G)$, the derivation in w takes less than k steps, and both the rank and fan-out of G are less than k.

The first problem is proven to be W[1]-hard already for $c = 2$, again by a reduction from the k-clique problem. There is currently nothing to suggest that this problem is *in* W[1], and unfortunately W[1] is likely already rather hard.

The second problem is proven to be W[SAT]-hard already for rank 1, by reduction from a type of satisfiability problem that is W[SAT]-complete. This is (most likely) an even harder class than the previous parameterization, and mostly serves to illustrate the need for a better choice of parameter. This also makes the third problem W[SAT]-hard, since if fixing the rank to one gives W[SAT]-hardness it cannot help us to include it in the parameter, as it then only goes up by a constant in infinitely many hard cases.

Finally, consider the fourth problem, where the rank, the fan-out, and the derivation length are all included in the parameter. As we keep adding more and more of the problem to the parameter the complexity of the resulting parameterized problem should hopefully fall (up until the $(p,|p|)$ case discussed above), the limitation on the derivation length limits the length of the strings possible, but still does nothing to control the overall size of the grammar. However, the proof of W[1]-hardness for the first problem type is here reapplied, as the reduction incidentally also constructs a grammar where all derivations are short (in the overall size of the grammar). Luckily it can be proven that this problem is *in* W[1]. This is proven by using a special case of parsing short derivations in context-sensitive grammars, which is known to be W[1]-complete, and applying this to our short LCFRS derivations through a careful FPT reduction.

[2] It may seem more logical to make k the sum or product of the rank and fan-out, but since all treatment of the parameter is through arbitrary computable functions this is unnecessary, as for example squaring the maximum of the rank and the fan-out is necessarily greater or equal to the product of the two.

CHAPTER 4

Constraining Language Concatenation

In this chapter we consider another operator which in some ways operates in the opposite way of the binary shuffle operator. The binary shuffle operator for two languages L and L' constructs a language $L \odot L'$ which is a superset of the concatenation $L \cdot L'$. This superset is created by, in a sense, softening the requirement of the concatenation point, and letting strings interleave into each other. Here we will introduce the *cut* operator, which creates a *subset* of $L \cdot L'$, which contains all of the concatenations for which it is not in any way *ambiguous* where one string ends and the next starts. This is quickly clarified once we leap into the definition.

We will in addition compare and contrast this type of operator with a number of features and properties of real-world regular expression engines. This chapter, since it deals with a somewhat singular practical regular expression feature, does not have a *-marked section, and instead has a slightly more technical slant in various parts. Familiarity with regular expressions is important for understanding the material here presented.

4.1 The Binary Cut Operator

The cut operator is a kind of concatenation of languages. To state this definition we need some additional notation. For any string $\alpha_1 \cdots \alpha_n \in \Sigma^*$ (i.e., $\alpha_i \in \Sigma$ for each i) let $\mathit{prefix}(\alpha_1 \cdots \alpha_n) = \{\alpha_1, \alpha_1\alpha_2, \ldots, \alpha_1 \cdots \alpha_n\}$, that is, all *non-empty* prefixes of $\alpha_1 \cdots \alpha_n$. Let $\mathcal{P}(S)$ denote the power-set of a set S. Then the definition of the cut is as follows.

Definition 4.1 (Binary Cut Operator) Let $! \colon \mathcal{P}(\Sigma^*) \times \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ be a binary operator such that

$$\mathcal{L} \,!\, \mathcal{L}' = \{uv \mid u \in \mathcal{L}, v \in \mathcal{L}', uv' \notin \mathcal{L} \text{ for all } v' \in \mathit{prefix}(v)\}$$

for any languages $\mathcal{L}, \mathcal{L}' \in \Sigma^*$. ◇

Notice that this definition ensures that $\mathcal{L} \,!\, \mathcal{L}' \subseteq \mathcal{L} \cdot \mathcal{L}'$, that is, the cut is a subset of the concatenation. The inclusion is not necessarily strict, for example if $\mathcal{L}' = \{\varepsilon\}$ it necessarily follows that $\mathcal{L} \,!\, \mathcal{L}' = \mathcal{L} \cdot \mathcal{L}'$.

Let us look at a series of examples (partially borrowed from Paper III, see Section 2 of that paper for further examples).

Example 4.2 (Empty Cuts) Let $\mathcal{L} = \{ab, abb, abbb, \ldots\}$ and $\mathcal{L}' = \{bc, bbc, bbbc, \ldots\}$ (that is, $\mathcal{L} = abb^*$ and $\mathcal{L}' = bb^*c$). Then $\mathcal{L} \,!\, \mathcal{L}'$ is the empty language. Let us consider one of the strings in the concatenation, *abbc*. This string cannot be in $\mathcal{L} \,!\, \mathcal{L}'$, as, looking at Definition 4.1, splitting it into $u = ab$ and $v = bc$, while fulfilling $u \in \mathcal{L}$ and $v \in \mathcal{L}'$, is not permitted as $b \in \mathit{prefix}(bc)$, and $abb \in \mathcal{L}$. Picking $u = abb$ leaves $v = c$, which is not in $\mathcal{L}'$. ◇

A more interesting language generated by a cut is the following.

Example 4.3 (Very Unsymmetrical Cuts) Let $\mathcal{L} = \{a, b, aa, bb, aaa, bbb, \ldots\}$ and let $\mathcal{L}' = \{ac, bc\}$. Then $\mathcal{L} \,!\, \mathcal{L}' = \{abc, bac, aabc, bbac, aaabc, bbbac, \ldots\}$. The reason is simple; if the u (in the sense of Definition 4.1) is chosen to be some number of "a"s, then v cannot be picked to be *ab*, since the *a* prefix will be consumed by $\mathcal{L}$. Similarly, if the string starts with a *b* it becomes impossible to pick v as *bc*.

This illustrates that while the cut $\mathcal{L} \,!\, \mathcal{L}'$ produces a subset of $\mathcal{L} \cdot \mathcal{L}'$ it does *not necessarily* produce a language of the form $L \cdot L'$ for some $L \subseteq \mathcal{L}$ and $L' \subseteq \mathcal{L}'$. ◇

4.2 Reasoning About the Cut

As a short aside, let us consider how the cuts relate to the other formalisms presented here. Where the shuffle operator effectively takes two strings, let us call them w and v, and interleaves them in such a way that, reading the result left to right, we (in general) have no idea which string each symbol belongs to. Perhaps v has not even started yet, or perhaps all of it has already been seen. The cut, on the other hand, is such that it *only permits* the concatenations where there is *no* ambiguity about where w ends. The only way wv is in the language generated by the cut is if the point of concatenation is decided entirely by the language w belongs to. That is, reading a string from a cut language from left to right we know that we have finished reading the w part when it is no longer possible for w to be longer. Then the remaining string must be the v part.

It is in this way the cut and the shuffle can be viewed as opposite directions from the concatenation, where the shuffle permits more ways of combining w and v, and the cut permits only a subset of all possible concatenations based on removing ambiguity when reading from the left.

4.3 Real-World Cut-Like Behavior

In the case of the cuts the real-world motivation is rather immediate and, hopefully, compelling. Regular expressions are a very popular tool for programmers, and regular languages of other forms also show up with great frequency. However, in mixing non-deterministic constructions for testing language membership and the deterministic flow control of the "main" program some very interesting effects can be achieved (or, alternatively, unexpected problems may be created, as the case may be).

Consider the Python function shown below, which successively matches multiple regular expressions to the same string.

Listing 4.4 (A Repeated Regular Expression Python Program)

```
# match argument against successive regular expressions
def matchx(s):
    # match a prefix of s against aa*|bb*
    m1 = re.match("^aa*|bb*", s)
    if m1 != None:
        # if ok, match the remainder of s against ab|bc
        m2 = re.match("ab|bc$", s[m1.end(0):])
        if m2 != None:
            # if both succeded report success
            return "Matched"
    # otherwise failure
    return "Did not match"
```

Basically, the code in Listing 4.4 is a function, which takes a string s as input, and then matches a *prefix* of s to the regular expression $aa^*|bb^*$ (the language $\{a,b,aa,bb,aaa,bbb,\ldots\}$), if that match is successful the *remaining suffix*, that is, whatever remains after removing the prefix that the first regular expression matched, of s is matched against the regular expression $ab|bc$ (the language $\{ab,bc\}$). If that match is successful success is reported.

The language "matched" by this program is exactly the language $(aa^*|bb^*)\,!\,(ab|bc)$. It might be easy to think that is *should* match $(aa^*|bb^*)\cdot(ab|bc)$ (which includes e.g. *aab* and *bbc*, which are not included in Example 4.3), but this is not the case. The thinking is exactly the one discussed for the cuts, the deterministic behavior of the outer program lets the first regular expression read as much as it wants, and the default behavior or regular expressions in most software libraries is to make the longest possible match. Once that has happened the suffix has been deterministically selected, and the second regular expression *must* match whatever is left for the overall match to work. In comparison, $(aa^*|bb^*)\cdot(ab|bc)$ features the non-deterministic behavior "intended" in regular expressions, the default behavior in most software packages will still be that the first part should match as much as possible, but if the overall match fails it will backtrack and choose a smaller match (if possible) in deference to the entire expression succeeding.

4.4 Regular Expressions With Cut Operators Remain Regular

4.4.1 Constructing Regular Grammars for Cut Expressions

Next we in short recap a result given in full in Paper III, showing that adding the cut operator to regular languages (or, of course, regular expressions) creates regular languages. We will, with some further extensions later, call these expressions which add

the cut to the normal set of regular expression operators *cut expressions*. The straightforward way to demonstrate that cuts preserve regularity is by employing a product construction, a variation of which was already employed in Section 2.4 to demonstrate the regularity of regular expressions extended with the binary shuffle operator.

Assume that we have some regular grammars (in the vein of Figure 1.3) R_1 and R_2, and that we wish to construct a *regular* grammar R for $R_1 \,!\, R_2$. Basically we will for each non-terminal A_1 in R_1 and each non-terminal A_2 in R_2 construct the non-terminals $(A_1, \perp)$, $(\perp, A_2)$ and (A_1, A_2) in the new grammar. The extra symbol $\perp$ is intended to signify "absent" here, and the new grammar will start out in $(S, \perp)$ where S is the initial non-terminal from R_1. The full construction then carefully ensures that whenever we have read a prefix of the input-string that R_1 *could* accept it starts R_2 on *its* initial non-terminal (i.e., if we are in (A, B) and R_1 can get rid of A without reading any more we go to (A, S'), where S' is the initial non-terminal for R_2). That is, as the string is read whenever R_1 can accept the string it restarts R_2 in its initial state, whenever R_2 can accept the grammar for $R_1 \,!\, R_2$ can accept.

The basic intuition behind this construction is that for every prefix that R_1 can accept Definition 4.1 says two things:

- It is *possible* that R_2 may start matching at this point, if R_1 cannot go on to match something longer.
- It is not allowed that we have *already* switched to matching in R_2.

In effect the construction speculatively keeps track of both R_1 and R_2, ensuring that R_1 gets its longest possible prefix of the string read, while keeping track of what R_2 has otherwise done.

The construction is hard to further simplify in a form that is more instructive than the full version, so refer to Lemma 2 of Paper III for a deeper explanation.

4.4.2 Potential Exponential Blow-Up in the Construction

While the cut expressions generate only regular languages, proving no more powerful than regular expressions, this does not mean that it is a pointless formalisms. There are two additional considerations to make.

1. Does the formalism allow something important to be conveniently expressed?
2. Does it express some languages in a more compact way?

In the case of cut expressions both are true. Modeling the loss of non-determinism illustrated in Listing 4.4 (and later in Listing 4.7) is interesting, and as we will demonstrate next there are also some families of languages where the smallest regular expression is exponentially larger than the equivalent cut expression, even when restricting ourselves to use only a single binary cut operator.

The core of this argument is simply that the cut can express a set difference of sorts on languages.

Lemma 4.5 Let $\Gamma = \Sigma \cup \{\#\}$, we assume that $\# \notin \Sigma$. Then $\left((L\#\Gamma^*) \,|\, \varepsilon\right) \,!\, (\Sigma^*\#) = (\Sigma^* \setminus L)\#$ for all languages L over Σ. ◇

A complete proof of this lemma is out of the scope of this introduction, but it is fairly intuitive when one considers the cases. Assume that $w \in L$, then, for the lemma to hold, $w\#$ should *not* be in $((L\#\Gamma^*)\,|\,\varepsilon)\,!\,\Sigma^*\#$. This is clearly the case, as the $L\#\Gamma^*$ will consume it entirely, leaving nothing for the trailing $\Sigma^*\#$ to match. This in fact holds for any string v with $w\#$ as a prefix, as the Γ^* keeps consuming all symbols. In the other direction, assume that $w \notin L$. Then $w\#$ is not matched by $L\#\Gamma^*$, meaning that the ε part of the branch is chosen, and then $\Sigma^*\#$ matches it and the match succeeds.

To exploit Lemma 4.5 we can now construct a *regular* expression R over Σ such that the shortest string R does *not* match is exponential in length (compared to the length of R). We can then apply Lemma 4.5, taking L as $\mathcal{L}(R)$, to produce a cut expression for which the shortest matching string is exponential in the length of the expression. From this we can then draw the conclusion that the smallest regular grammar (or finite automaton or regular expression) is at least exponentially larger than the cut expression. Recall Definition 1.6 in preparation, and note that R^k (for some regular expression R and $k \in \mathbb{N}$) is not a regular expression operator, but is here used as an abbreviation for

$$\underbrace{R \cdot R \cdots R}_{k \text{ times}}.$$

If regular expressions are extended with an actual R^k operator the explosion in size would be even greater.

We select $\Sigma = \{0, 1, \$\}$ as the alphabet, and let $\Delta = \{0, 1\}$. For each $n \in \mathbb{N}$ the regular expression R_n has five components, $R_n = A\,|\,B\,|\,C_n\,|\,D_n\,|\,E_n$, which are as follows. In the end the language considered will be $(\Sigma^* \setminus \mathcal{L}(R_n))\#$, so the aspect to consider is e.g. the language $\Sigma^* \setminus A$ and so on.

1. $A = \Delta\Sigma^*\,|\,\$\Delta^*1\Delta^*\$\Sigma^*$. Note that all strings in $\Sigma^* \setminus A$ start with \$ and contain no 1 until the next \$.

2. $B = \Sigma^*\Delta\,|\,\Sigma^*\$\Delta^*0\Delta^*\$$. Note that all strings in $\Sigma^* \setminus B$ end with \$ and contain no zero between the last two \$ symbols.

3. $C_n = \Sigma^*\$\Delta^{n+1}\Delta^*\$\Sigma^*\,|\,C_{n,0}\cdots C_{n,n-1}$ where $C_{n,i} = \Sigma^*\$\Delta^i\$\Sigma^*$ for each i. Note that all strings in $\Sigma^* \setminus C_n$ have exactly n zeroes and ones between each pair of \$ symbols.

4. $D_n = D_{n,1}\,|\cdots|\,D_{n,n-2}$, where $D_{n,i} = \Sigma^*\$\Delta^i0\Delta^*0\Delta^*\$\Delta^i1\Delta^*\$\Sigma^*$ for each i. Note that the strings in $\Sigma^* \setminus D_n$ are such that every substring $\$x\$y\$$ (with $x, y \in \Delta^n$, which will be enforced by C_n), is such that if the ith symbol in x is a zero, and the ith symbol in y is a 1, then symbols $i+1$ through n in x must be ones.

5. $E_n = E_{n,1}\,|\cdots E_{n,n-2}$ where $E_{n,i} = \Sigma^*\$\Delta^i01^{n-i-1}\$\Delta^i1\Delta^*1\Delta^*\$\Sigma^*$ for each i. Note that all strings in $\Sigma^* \setminus E_n$ are such that every substring $\$x\$y\$$ (with $x, y \in \Delta^n$, which will be enforced by C_n), is such that if the ith symbol in x is a 0, symbols $i+1$ through n are ones, and the ith symbol in y is a 1, then the remainder of y must be zeroes.

Taking all these together we learn that each $w \in (\Sigma^* \setminus \mathcal{L}(R_n))$ are strings such that $w = \$x_1\$x_2\$\cdots\$x_m\$$ where each x_i is a string of n zeroes and ones (due to C_n), such that $x_1 = 0\cdots0$ (due to A), and $x_n = 1\cdots1$ (due to B). At most one zero in x_i can be turned into a one in x_{i+1}, and only if all the subsequent positions were ones in x_i and are zeroes in x_{i+1}. From this it directly follows that the shortest string in $\Sigma^* \setminus \mathcal{L}(R_n)$ will be the sequence of all n-bit binary strings in order, for example

$$\$000\$001\$010\$011\$100\$101\$110\$111\$ \in R_3.$$

In addition a number of *longer* strings exist, which show up since a one in x_i may turn into a zero in x_{i+1}. However, the only way to get from the initial zero sequence to the final one sequence is to increment by one in the binary addition sense at least 2^n times.

Notice, however, that the actual expression R_n is on the order of n^2 symbols long, where C_n, D_n and E_n are the big part. Applying Lemma 4.5 constructs a cut expression accepting $(\Sigma^* \setminus \mathcal{L}(R_n))\#$, which is still on the order of n^2 symbols long, but as argued above the shortest string it accepts is exponential in n.

Non-extended regular expressions, regular grammars (as sketched in figures here) and finite automata are all such that the shortest string they accept is at most linear in the size of the expression/grammar/automaton (if they accept any string at all). This is easy to see, some efficient shortest path algorithm can be employed to find a path through the expression/grammar/automaton. As such, cut expressions are exponentially more succinct in some cases, and converting an arbitrary cut expression into one of those listed formalisms may create an exponentially larger representation. This will be rather key to understanding the complexity of solving the membership problem.

4.5 The Iterated Cut

In a further parallel with Chapter 2 we also consider a unary iterated version of the cut. Much like $R^{\odot} = R \odot R \odot \cdots \odot R$ we let $R^{!*} = R \mathbin{!} (R \mathbin{!} (R \mathbin{!} \cdots (R \mathbin{!} R)\cdots))$. However, notice that while the shuffle operator is associative, the cut operator is not.

Example 4.6 Let us consider two expressions differing only in associativity $((ab)^* \mathbin{!} a) \mathbin{!} b$ and $(ab)^* \mathbin{!} (a \mathbin{!} b)$.

- For $((ab)^* \mathbin{!} a) \mathbin{!} b$ clearly $(ab)^* \mathbin{!} a$ is the same as $(ab)^*a$, since the $(ab)^*$ part cannot cover the final a, so $\mathcal{L}((ab)^* \mathbin{!} a) = \{a, aba, ababa, \ldots\}$, and, hence, $\mathcal{L}(((ab)^* \mathbin{!} a) \mathbin{!} b) = \{ab, abab, ababab, \ldots\}$.
- However, if we instead consider $(ab)^* \mathbin{!} (a \mathbin{!} b)$, we notice that $a \mathbin{!} b$ is the same as ab, so we have $(ab)^* \mathbin{!} ab$ which is clearly empty, as the $(ab)^*$ will consume all repetitions of ab ensuring the second part never gets to match anything.

It follows that the cut operator is *not* associative. ◇

The iterated cut, in a sense similar to how the binary cut models the program in Listing 4.4, permits the modeling of loops of regular expression matching, like in the below listing.

Listing 4.7 (A Looping Regular Expression Python Program)

```
# match s against the regular expression R repeatedly
def matchy(R, s):
    # keep matching
    while True:
        # match s to re once
        m = re.match(R, s)
        # if the match failed report failure
        if m == None:
            return "Did not match"
        # otherwise, extract the remainder of the match
        s = s[m.end(0):]
        # if the whole string matched, report success
        if len(s) == 0:
            return "Matched"
```

This listing will give the same behavior as trying to match s to $R^{!*}$.

The iterated cut is hard to express directly in a regular expression with just the addition of the binary cut operator. Notably it is not a matter of nesting cuts inside of Kleene closures, like $(R\,!\,R)^*$ or similar, as this will give too much non-deterministic freedom in general. However, adding both the binary cut operator *and* the iterated cut to regular expression *still* produces expressions that can only generate regular languages. The construction for this part is slightly trickier than for the case of the binary cut operators, so it is best to refer to Paper III where complete and formal constructions for both cases are given.

4.6 Regular Expression Extensions, Impact and Reality

4.6.1 Lifting Operators to the Sets

Recall Definition 1.6 where the basic operations in regular expressions are defined. It is an important fact to note that each of those classical regular expression operators are expressed string-wise. That is, an operator f takes n argument subexpressions $R_1, \ldots, R_n$, and the language it generates is then

$$\mathcal{L}(f(R_1, \ldots, R_n)) = \{f(v_1, \ldots, v_n) \mid v_1 \in \mathcal{L}(R_1), \ldots, v_n \in \mathcal{L}(R_n)\}.$$

That is, the classic operators all operate "point-wise" on strings, and this is then lifted to the level of sets (i.e. we can take the categorial view and consider a functor here) to generate languages. However, the cut does not operate on this level. Instead Definition 4.1 operates on the level of the language. We can talk about $L\,!\,L'$ for languages, but informed that $w \in L$ and $v \in L'$ we cannot from this determine whether $wv \in L\,!\,L'$.

This *should* be viewed as a flaw with the cuts, their introduction into expressions does change the nature of the expression in a fundamental way. On the other hand, the impact is comparatively small when contrasted to the cut-like operators that many

regular expression software packages include. These have behavior that is even further from the clean nature of the classical operators.

4.6.2 An Aside: Regular Expression Matching In Common Software

This way of phrasing how matching happens may appear unusual for anyone more familiar with the more classic regular expression constructions, where the semantics are described in a composed way, i.e., $\mathcal{L}(R_1|R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$, etc., or by constructing finite automata for the expression (constructing the Glushkov automaton [Glu61] and determinising it, or directly constructing a deterministic finite automaton using e.g. derivatives [Brz64]). Most practical software packages, however, use a depth-first backtracking search across some abstract syntax tree representation of the regular expression. The reasons for this are two-fold.

1. The efficiency of this approach is in many cases great. Constructing the syntax tree is efficient, and the representation is in general far more compact than the automata approach. The actual search may in the worst case be a lot slower (exponential in the length of the string), but the semantics are straightforward enough that the task of structuring the expression in a way that gives efficient matching in the most common cases can be left to the programmer.

2. It enables a multitude of additional regular expression features. Most immediately it makes it possible to deterministically talk about which part of the regular expression matches which part of the string. That is, $(a^*|b^*)(a|b)^*$ matches *aaababa*, but which part of the expression matches the *aaa* prefix? In the theoretical setting this is a nonsense question, all we state is that $aaababa \in \mathcal{L}$, the *how* is entirely undefined. In regular expression software packages however the initial three *a*s will be matched by the first a^*, and this information can be extracted with the API provided. Which parts of the expression will "prefer" to match what can be controlled further with a variety of operators, and the pieces of the string matched by a certain subexpression can even be recalled inside the expression (permitting the language $\{ww \mid w \in \Sigma^*\}$ to be matched by recalling a copy of the string already matched).

In short; the accepted approach has numerous implications for the functionality and performance of regular expression matching in practice.

4.6.3 Real-World Cut-Like Operators

There are a variety of operators in practical regular expression packages which behave *somewhat* similar to cuts. The first, and most common, are the possessive quantifiers. Let us look specifically at the possessive variation of the Kleene star R^* as defined in Definition 1.6, denoted R^{*+}. Defining the language generated by R^{*+} leads to disappointment however, $\mathcal{L}(R^{*+}) = \{\varepsilon\} \cup \{vw \mid v \in \mathcal{L}(R), w \in \mathcal{L}(R^{*+})\}$ inductively. Unfortunately this is precisely the same language as generated by $\mathcal{L}(R^*)$, which is because the possessive quantifier does not operate on the same level as classical regular operators, or even the set-level behavior of the cut operators. Instead the semantics of the possessive quantifiers are intertwined with the overall matching of the entire expression in a

way that is hard to formalize. Consider the examples in Table 4.8 which are produced using the Java (1.6.0u18) regular expression implementation. Notice how applying

Table 4.8: Some regular expressions using possessive quantifiers and the language they accept in Java 1.6.0u18.

Expression	Language
$(aa)^{*+}a$	$\{a, aaa, aaaaa, aaaaaaa, \ldots\}$
$((aa)^{*+}a)^*$	$\{\varepsilon, a, aaa, aaaaa, aaaaaaa, \ldots\}$
$((aa)^{*+}a)^*a$	$\{a\}$

the Kleene star to the expression in the first row does not (in the second row) generate for example *aa*, despite *a* being in the language of the first row.

We will not attempt to deeply explain the semantics of this operator, but it operates by manipulating the internal backtracking search. The outcome does not easily fit into the compositional classic explanation of how regular expressions generate languages. See Paper III for more examples of this type of operator.

As an addition, some regular expression engines feature an additional binary operator, `(*PRUNE)`, that compares fairly directly to the binary cut operator (in that it is not attached to a Kleene star), but still has semantics that are hard to comprehend form the compositional perspective. See Table 4.9 for some examples of expressions and the languages recognized in Perl 5.16.2.

Table 4.9: Some regular expressions using the `(*PRUNE)` operator and the language they accept in Perl 5.16.2, similar to the examples in Table 4.8.

Expression	Language
$(aa)^*$ `(*PRUNE)` a	$\{a, aaa, aaaaa, aaaaaaa, \ldots\}$
$((aa)^*$ `(*PRUNE)` $a)^*$	$\varnothing$
$((aa)^*$ `(*PRUNE)` $a)^*a$	$\varnothing$

4.6.4 Exploring Real-World Regular Expression Matchers

Paper IV explores the behavior of these practical software package matchers. They effectively operate by constructing an automaton (or grammar) where rules are prioritized, whenever there are multiple rules that *could* be applied there is a preferred rule that is tried first. If applying that rule does not lead to accepting the string the procedure backtracks and tries the other options. The full discussion requires a deep technical look at the behavior of the software, and is best explored by reading Paper IV. Suffice it to say, beyond exploring the semantics to analyze additional operators, this search procedure will additionally at times require exponential time. Consider for ex-

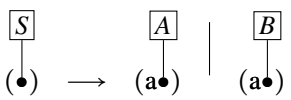
ample the expression $(a|a)^*$, trying to match the string $a\cdots ab$. It will fail to match the b, but in the process the matching procedure will pick whether the first or second a should match each a in the string, and when the failure on b happens the backtracking will attempt *every other* way of matching the as to the string. In fact, attempting this match using Java on the authors (reasonably modern) machine the runtimes shown in Table 4.10 are achieved. The main contribution in Paper IV is in statically analyzing

Table 4.10: The time in seconds it takes to match the string $a\cdots ab$ to the regular expression $(a|a)^*$ in Java on the authors desktop PC, as it depends on the number of as in the string. Notice the almost perfect power of two exponential growth.

Number of "a"s in $w = a\cdots ab$	23	24	25	26	27	$\cdots$	30
Seconds to match w **to** $(a|a)^*$	1.04	2.00	3.66	7.22	13.56	$\cdots$	118.81

regular expressions for this type of exponential worst-case behavior (i.e., a^* can never take exponential time, since there is only one choice, but $(a^*)^*$ can).

One additional point of interest in Paper IV is how the matcher picks which choice to explore first. This is done by giving finite automata priorities, where one choice is more prioritized than another. This leads to the definition of the prioritized non-deterministic finite automata (pNFA) formalism in Paper IV. These are fairly straightforward, if we imagine the rule



we now say that the first possibility, which generates the non-terminal A, is *prioritized*. That is, if A can generate the rest of the string we prefer to have it do so, and try generating the rest with B only if A fails. This distinction makes no difference for the language accepted, but it makes it unambiguous how the string is generated, which ensures that the solution to a parsing problem instance is unequivocal.

4.7 The Membership Problem for Cut Expressions

Parts of the membership problem for cut expressions should already be clear; namely, Section 4.4 and Section 4.5 together demonstrate that the cut expressions generate only regular languages. The *non-uniform* membership problem for regular languages is decidable in linear time, so we can decide the non-uniform membership problem for cut expressions in linear time, since we can just rewrite the cut expression into a regular grammar or similar (through the arguments in the aforementioned sections).

However, as Section 4.4.2 demonstrates, the regular grammar may be exponentially large, so the equivalence to regular grammars gives us no more than an exponential algorithm for deciding the uniform membership problem. Luckily a very

direct table parsing algorithm can decide membership in cubic time. Let us sketch very quickly how it is done.

Algorithm 4.11 (Parsing for Cut Expressions) Take as input a cut expression E and a string $\alpha_1 \cdots \alpha_n$. Let S_E denote the set of subexpressions of E (including E itself).

1: Construct the table $T : S_E \times \{1,\ldots,n+1\} \times \{1,\ldots,n+1\} \to \{true, false\}$.
2: Set $T(E,i,j) := false$ for all E,i,j at the start
3: **for** $S \in S_E$, working bottom-up through the sub-expressions **do**
4: **if** $S = \varepsilon$ **then** $T(S,i,i) := true$
5: **else if** $S \in \Sigma$ **then** $T(S,i,i+1) := true$ for all i with $\alpha_i = S$
6: **else if** $S = E_1 \,|\, E_2$ **then**
7: $T(S,i,j) := true$ for all $i \leq j$ s.t. $T(E_1,i,j) \vee T(E_2,i,j)$
8: **else if** $S = E_1 \cdot E_2$ **then**
9: $T(S,i,k) := true$ for all $i \leq j \leq k$ s.t. $T(E_1,i,j) \wedge T(E_2,j,k)$
10: **else if** $S = E_1^*$ **then**
11: $T(S,i_1,i_n) := true$ for all n, $i_1 \leq \cdots \leq i_n$ s.t. $T(E_1,i_1,i_2) \wedge \cdots \wedge T(E_1,i_{n-1},i_n)$
12: **else if** $S = E_1 \,!\, E_2$ **then**
13: $T(S,i,k)$ for all $i \leq j \leq k$ such that:
14: $T(E_1,i,j) \wedge T(E_2,j,k)$, and,
15: $\neg T(E_1,i,j')$ for all $j < j' \leq k$.
16: **end if**
17: **end for**

This algorithm is trivially cubic (quadratic in the length of the string), since every table position is set true at most once. The case for the shuffle closure is not included, but is a trivial addition.

After the threat of potentially exponentially large regular grammars the cubic time (and space) of the above algorithm may be calming, but given the typical efficiency of matching classical regular expressions cubic time is still not entirely pleasing. Better algorithms remain an open question however, very notably Section 4.4.2 demonstrates a case where applying a cut exponentially blows up the size of the smallest corresponding regular grammar exponentially, but for the upper bound we only know that it cannot be *worse* than non-elementary, which is not very satisfying. This in fact follows from the product-style construction discussed in Section 4.4.1, and is discussed at greater length in Paper III.

CHAPTER 5
Block Movement Reordering

This short chapter discusses matters of block reordering, which is once again a non-obvious term, this time lifted form the field of edit distance, where operations that modify multiple symbols in a contiguous substring at once are referred to as block operations. Specifically the topic of interest is attempting to study the results of reordering nodes in the parse tree for a string, which gives rise to a sort of hierarchical block movement reordering in the underlying string language.

5.1 String Edit Distance

String edit distance is a long studied field. It is concerned with defining a distance between strings using a sequence of operations (reminiscent of the rule-based derivations discussed in earlier chapters, but starting from another, possibly *longer*, string). The distance measure is defined in terms of a set of operations, each of which makes some small modification to a string, and then the distance between a string $w \in \Sigma^*$ and $v \in \Sigma^*$ is the minimum number of operations (possibly weighted in some way) we need to apply to modify w into v. The problem of finding this sequence is known as the *string correction problem*. A classic set of operations for this is to have an operation to *delete* a single symbol and one to *insert* a single symbol. Making the distance from e.g. *abc* to *cca* four, since the initial *ab* must be removed (two removal operations) and *ca* must be added at the end. A typical additions to the set of operations is to add an operator to *replace* one symbol by another, this set of three operators is called Levenshtein distance [Lev66]. The next typical addition, and most important for us here, is the *swap*, which swaps the positions of two adjacent symbols, the resulting set of operators is called Damerau-Levenshtein distance [Dam64].

5.2 A Look at Error-Dilating a Language

The direction of interest here starts out from the question of an error *dilation* of a language. Consider Figure 5.1. That is, we choose a language class $\mathcal{G}$ (perhaps the regular or context-free languages) and a string edit distance e, then for each language $L \in \mathcal{G}$ and each $k \in \mathbb{N}$ we define $L_{e=k}$ to mean that $w \in L_{e=k}$ if and only if there exists some $v \in L$ such that w is k or less distance from v. Notably, as k approaches ∞ the

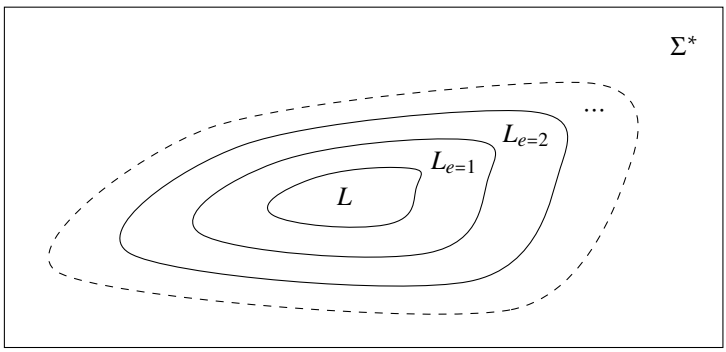


Figure 5.1: A diagram of the dilation of a language through error measures.

language $L_{e=k}$ approaches Σ^*.

Performing such a construction is fairly straightforward for most choices of formalisms. If we consider just the case of *insert* and *delete* with a regular grammar G, and the constant k chosen, then we can for each non-terminal A construct $k+1$ new non-terminals $A_k, A_{k-1}, \ldots, A_0$. The non-terminal A_i has all the rules that A would have, with i preserved, so for example the left rule turns into the right in the following way.

$$\boxed{A} \to (\bullet) \longrightarrow \boxed{B} \to (\alpha\bullet) \qquad \boxed{A_i} \to (\bullet) \longrightarrow \boxed{B_i} \to (\alpha\bullet)$$

In addition for each $\alpha \in \Sigma$ and non-terminal A_i with $i > 0$ we add the following rule.

$$\boxed{A_i} \to (\bullet) \longrightarrow \boxed{A_{i-1}} \to (\alpha\bullet)$$

This allows one "insertion" to be used, we count down the number allowed and add an arbitrary symbol.

Finally, for each existing rule that *would* add a symbol we simulate a deletion by adding a rule that counts down i but "fails" to generate the symbol, as above with the left original rule and the right new rule (though one for each $0 < i \leq k$ must be generated of course).

$$\boxed{A} \to (\bullet) \longrightarrow \boxed{B} \to (\alpha\bullet) \qquad \boxed{A_i} \to (\bullet) \longrightarrow \boxed{B_{i-1}} \to (\bullet)$$

Finally, we let the starting non-terminal S go to S_k (to signify that we start out with k operations available). Notice how, once the subscript gets to zero, only the "original" rules are usable, each use of a insertion/deletion rule "costs" one from the subscript.

5.3 Adding Reordering

5.3.1 Reordering Through Symbol Swaps

Adding the simple symbol swap to the prior construction is only minimally more complex. We can extend the tagging of the non-terminals to remember "we pretended to swap α for a β", meaning that we generated a β from a rule that should have generated α, and this tagging of the non-terminal lets it only take rules which have been modified such that they *originally* generated β, but in this modified rule they generate the missing α and the derivation continues on as normal.

5.3.2 Derivation-Level Reordering

We now get to the real aim of this section, the intent of this edit distance is to model some sort of error or imprecision, however, in the context of a lot of languages simply replacing symbols may not really reflect the nature of errors properly. Consider for example in natural languages, where large grammatical restructurings may be only "slightly" bad, since they still obey some basic rules. That is, "She chases a blue ball" is correct English, whereas "A blue ball she chases" is slightly ungrammatical, but still completely understandable, whereas "ball she chases a blue" is incomprehensible despite involving less reordering.

We have so far dodged the issue of parse trees, but here they become rather core to the question. Consider Figure 5.2. This illustrates (a possible interpretation of)

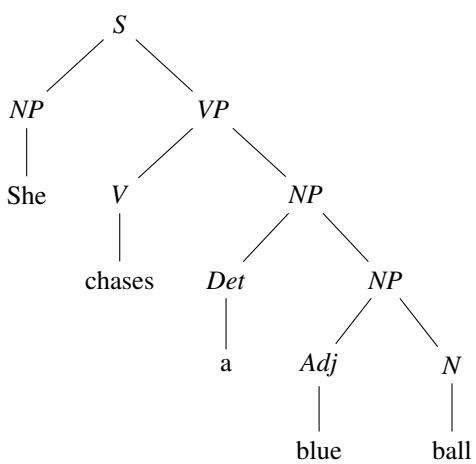


Figure 5.2: A parse tree for the sentence "She chases a blue ball", the internal nodes of the tree corresponds to the non-terminals which generated that part of the sentence, the words in the leafs are the symbols of Σ in this case.

the structure of the natural language sentence. It stands to reason that small modifications of this tree will have a closer relationship to the original sentence than small modifications to the string which forms the sentence.

5.3.3 Tree Edit Distance

Tree edit distance is a natural way to think about this, that is, we would like to create an error dilation of a language, in the style of Figure 5.1 using tree operations on the parse tree (which requires a specific instance of a grammar for the language to make sense) to modify the final strings.

The problem of tree edit distance is fairly well explored in some limited settings, see e.g. [Sel77, Tai79]. This work has for the most part however been constrained to just allowing insert and delete operators, the swap, or similar subtree movement operators, is a trickier matter [ZS89, Kle98, Bil05]. This is partially necessary, the tree edit distance on unordered trees (i.e., we allow deletions and insertions of nodes, but siblings in the tree have no order) is NP-complete [ZSS92]. We can simulate the unordered case if swaps are permitted, by simply replacing each internal node by a long chain of copies of the node. This way the swap remains cheap (it does not care how many nodes it moves in swapping two siblings) while making insertions and deletions expensive. If we add sufficiently many of these nodes the result will be that all orders can be achieved cheaper than it is to perform a single insertion or deletion, effectively making the problem behave like the unordered case.

5.4 Analyzing the Reordering Error Measure☆

Paper VI considers the very limited case of *only* permitting tree swaps in the distance measure, each swap having a cost of one. Let us consider the proper definition, recalling the definitions of trees from Section 2.9.1. First we define the swap distance between permutations.

Definition 5.3 Let $\pi_n \subset \mathbb{N}^n$ denote the set of permutations of length n, that is, $p_1 \cdots p_n \in \pi_n$ if and only if $\{p_1, \ldots, p_n\} = \{1, \ldots, n\}$. Then $p_1 \cdots p_n \in \pi_n$ has a swap distance less than or equal to k, denoted $swap(p_1 \cdots p_n, k)$ if and only if

- $k \geq 0$ and $p_1 \cdots p_n = 1 \cdots n$,
- there exists some i such that $swap(p_1 \cdots p_{i-1} p_{i+1} p_i p_{i+2} \cdots p_n, k-1)$. ◇

Then the tree variant is as follows.

Definition 5.4 For two trees t, t' we say that t and t' are within tree swap distance k, denoted $swap(t, t', k)$, if and only if $t = \alpha[t_1, \ldots, t_n]$ and $t' = \alpha[t'_1, \ldots, t'_n]$ for some $\alpha \in \Sigma$ and n, and there exists some $p_1 \cdots p_n \in \pi_n$ and $l_0, \ldots, l_n \in \mathbb{N}$ such that

- $k \geq \sum_{i=0}^{n} l_i$,
- $swap(p_1 \cdots p_n, l_0)$, and
- $swap(t_i, t_{p_i}, l_i)$ for all i.

The triple (t, t', k) is a "yes" instance of the tree swap distance problem if and only if $swap(t, t', k)$. ◇

Unfortunately this problem is proven to be NP-hard in Paper VI (the problem is obviously *in* NP, since the permutations for each level of the tree can be guessed and then verified in polynomial time). Let us briefly outline the process. The reduction starts with the *extended string-to-string correction problem*, which is an edit distance with *only* delete and swap operations[1]. This problem is known to be NP-complete (see e.g. [GJ90]). The reduction makes an intermediary stop in a problem that may be interesting in itself.

Definition 5.5 Let $M : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be an n by n matrix (i.e. a matrix with n rows and columns, letting $M(i, j)$ denote the value at the ith row and jth column), then (M, k) is a "yes" instance of the swap assignment problem if and only if there exists a permutation $p_1 \cdots p_n \in \pi_n$ and $l \in \mathbb{N}$ such that

- $swap(p_1 \cdots p_n, l)$
- $k \geq l + \sum_{i=1}^{n} M(p_i, i)$ ◇

That is, the problem is to decide whether it is possible to swap rows positions in M in such a way that the sum of the number of swaps used and the diagonal in the resulting matrix is less than or equal to k. The reduction is such that e.g. position $M(i, j)$ is zero if symbol i of the first string in the original edit distance problem is the same as symbol j in the second, combined with some trickery to enable deletions.

This matrix problem is then in turn reduced to the tree swap problem of Definition 5.4. This reduction is not overly difficult, the tree constructed will have height 3, the root has n immediate children corresponding to the rows, these have n children corresponding to the column positions in that row, and finally these have a coding of the number they should contain as children. Everything is distinctly coded so the swaps can only be used to reorder the rows, and to make the binary representations equal (which costs exactly the absolute difference between the numbers).

The fact that the tree swap distance is NP-complete is unfortunate, however, the amount of distance permitted in the error-dilating of languages should be very constrained (e.g. a sentence with three or more errors will often be incomprehensible already), so fixed parameter analysis and other more nuanced analysis would be of great interest.

[1] This statement abuses the notion of a distance heavily, since it is asymmetric. It does however fall into a similar class of problems.

CHAPTER 6
Summary and Loose Ends

None of the matters here can be considered settled or treated with some deep finality. This is a snapshot of ongoing research, here tied together with an overarching theme, but it is both likely and desirable that everything here treated will be supplanted with new greater results in the future. As such this concluding chapter attempts to look forward, while and noting the missing pieces, as well as summarizing some of the aspects of the attached papers that have not yet been brought up.

6.1 Open Questions and Future Directions

6.1.1 Shuffle Questions

There are two open questions from the preceding licentiate thesis [Ber12] that may be interesting to recall.

1. Deciding the membership problem for the shuffle of palindromes:

$$\{ww^{\mathcal{R}} \mid w \text{ is any string, } w^{\mathcal{R}} \text{ is } w \text{ reversed}\}.$$

2. Deciding the membership problem for the language of shuffle squares,

$$\{w \odot w \mid w \text{ is any string}\}.$$

Notice that as the languages we are concerned with are specified as part of the problem these should be viewed as non-uniform membership problems.

The first remains a point of interest, Paper I demonstrates that the non-uniform membership problem for two linear deterministic context-free languages is NP-hard (see Chapter 2), and the shuffle of two palindromes seems like, in a spirit rather similar to the ideas of Paper V, or possibly more illustratively the Chomsky-Schützenberger theorem [CS63], the next step. That is, the palindromes are sort of the most primitive representation of the basic power that differentiates the linear deterministic context-free languages from the regular, in that both the intersection and homomorphism in the Chomsky-Schützenberger decomposition of it do "nothing". The author has no speculation whether this problem should be expected to be in P or not.

The shuffle square, on the other hand, has seen some important developments since [Ber12], and is proven NP-complete in a rather tricky reduction in [BS13].

In addition, let us note that the problems as stated above deal with languages with arbitrarily large alphabets (i.e., when it says that w is any string it may be over an alphabet up to $|w|$ in size). The reduction in [BS13] works for a finite alphabet version of the shuffle square as well, meaning that the language is NP-complete either way. No results are known for the palindrome shuffle, so a possibility is that the problem is NP-complete for arbitrarily large alphabets, but is in P for all alphabet sizes smaller than some constant.

Beyond that, there are numerous additional problems that may be considered in shuffle, especially as many aspects are of practical interest. Beyond simply improving on many of the results in Paper I, and considering both more generalized and restricted cases (shuffle on trajectories is a lively and interesting case), the problem the author most wants to highlight is the one considered in Paper V. That is, proving that for all context-free languages $L \subseteq \Sigma^*$ and $L' \subseteq \Gamma^*$ (with $\Sigma \cap \Gamma = \varnothing$) the shuffle $L \odot L'$ is context-free if and only if one of L and L' is regular.

6.1.2 Synchronized Substrings Questions

The synchronized substrings formalisms, such as linear context-free rewriting systems, are a prime example of where the details of parsing complexity are hugely important. The uniform membership algorithm appears inefficient from the classical complexity theory perspective, but in practice the algorithms are considered reasonably efficient (recall Section 3.4.3). Paper II does find some potentially efficient cases, but they are not necessarily entirely satisfactory, as the one truly efficient case identified is where the rank, fan-out *and* derivation length are included in the parameter (i.e., if all three are small the parsing problem is efficient).

The most obvious case not yet studied is to take the opposite approach from the classical non-uniform membership problem[1]; we let the length of the string be the parameter and consider the grammar in full, or near full. To see the reasoning here the intended application may need to be clarified. These formalisms are typically used for natural language processing. In this case it is easy to see that the sizes of the components are backwards from what is usually assumed, the strings are simply natural language sentences, and, while they can be long, like this run-on sentence, there are still very real practical limits on how many words there can be in one. A *reasonably complete* grammar for English however is vast at the best of times, simply enumerating exceptions will create tens of thousands of rules. As such the complexity in the grammar is actually more important than the complexity in the string.

6.1.3 Regular Expression Questions

The two Papers III and IV both deal with very similar issues, in that they are motivated by the order-dependencies that exist in practical regular expression semantics as an effect of matching methodology employed. Their approaches are very different how-

[1] Notice however, here we talk about parameterized complexity, the intent is not to clumsily assume some parts constant like in the non-uniform case. Parameterized complexity bounds still take the parts put in the "parameter" into account, but differentiate between how large a role that part plays in the complexity. See Section 3.5.1.

ever, in that Paper III attempts to bring an approximation (attempting to make them behave nicely within the classical framework) of these effects into a formal framework, whereas Paper IV tries to analyze the actual state of being of these regular expression engines using formal techniques. The way forward here is not immediately obvious, there are clear open questions that follow directly from Paper III (e.g. an upper bound on the automaton size), as well as some mechanical improvements already considered. On the other hand Paper IV having a continuation is to a great extent a question of impact, as the paper may very well inspire changes in regular expression engines, which would make continued research chase a moving target. A possibility which has both advantages and disadvantages.

As such there is a wealth of possible work in the area of regular expression semantics, but beyond incremental open questions which are already listed in the papers themselves this direction depends on the expected and actual impact of the research.

6.1.4 Other Questions

Paper V is obviously a work in progress published primarily for inclusion in this thesis. The conjecture presented does, however, appear very promising and significant, far beyond proving the open question for context-free shuffles discussed above. In the other direction, Paper VI is the oldest paper included, and is concerned with a direction that has not gotten a high level of attention from the author since. Continuing work appears to be a matter of extending the discussion in Chapter 5 in a way that arrives at a reasonably compelling language class, while having clearly motivated fixed parameter complexity problem with a positive outcome.

6.2 Conclusion

As a final remark the author wishes to again thank all his collaborators and colleagues, as well as everyone who worked on the many pieces of research leveraged as preliminaries in this work. Finally, the author thanks the reader for the interest shown.

References

[Bar85] G. Edward Barton. On the complexity of ID/LP parsing 1. *Computational Linguistics*, 11(4):205–218, 1985.

[BBB13] Martin Berglund, Henrik Björklund, and Johanna Björklund. Shuffled languages – representation and recognition. *Theoretical Computer Science*, 489-490:1–20, 2013.

[BBD13a] Martin Berglund, Henrik Björklund, and Frank Drewes. On the parameterized complexity of Linear Context-Free Rewriting Systems. In *Proceedings of the 13th Meeting on the Mathematics of Language (MoL 13)*, pages 21–29, Sofia, Bulgaria, August 2013. Association for Computational Linguistics.

[BBD+13b] Martin Berglund, Henrik Björklund, Frank Drewes, Brink van der Merwe, and Bruce Watson. Cuts in regular expressions. In Marie-Pierre Béal and Olivier Carton, editors, *Proceeding of the 17th International Conference on Developments in Language Theory (DLT 2013)*, pages 70–81, 2013.

[BDvdM14] Martin Berglund, Frank Drewes, and Brink van der Merwe. Analyzing catastrophic backtracking behavior in practical regular expression matching. Submitted to the 14th International Conference on Automata and Formal Languages (AFL 2014), 2014.

[Ber11] Martin Berglund. Analyzing edit distance on trees: Tree swap distance is intractable. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2011*, pages 59–73. Prague Stringology Club, Czech Technical University, 2011.

[Ber12] Martin Berglund. *Complexities of Parsing in the Presence of Reordering*. Licentiate thesis, Umeå University, 2012.

[Ber14] Martin Berglund. Characterizing non-regularity. Technical Report UMINF 14.12, Computing Science, Umeå University, `http://www8.cs.umu.se/research/uminf/`, 2014. In collaboration with Henrik Björklund and Frank Drewes.

[Bil05] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

[BN01] Eberhard Bertsch and Mark-Jan Nederhof. On the complexity of some extensions of rcg parsing. In *IWPT*, 2001.

[Bou98] Pierre Boullier. Proposal for a Natural Language Processing Syntactic Backbone. Research Report RR-3342, INRIA, 1998.

[Bou04] Pierre Boullier. *Range concatenation grammars*, pages 269–289. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[Brz64] Janusz Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[BS13] Sam Buss and Michael Soltys. Unshuffling a square is np-hard. *Journal of Computer and System Sciences*, 2013.

[CS63] Noam Chomsky and Marcel Paul Schützenberger. The Algebraic Theory of Context-Free Languages. In P. Braffort and D. Hirshberg, editors, *Computer Programming and Formal Systems*, Studies in Logic, pages 118–161. North-Holland Publishing, Amsterdam, 1963.

[Dam64] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, 1964.

[DHK97] Frank Drewes, Annegret Habel, and Hans-Jörg Kreowski. Hyperedge replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*, chapter 2, pages 95–162. World Scientific, 1997.

[EB98] Zoltan Ésik and Michael Bertol. Nonfinite axiomatizability of the equational theory of shuffle. *Acta Informatica*, 35(6):505–539, 1998.

[FG06] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Springer-Verlag, 2006.

[Gaz88] Gerald Gazdar. Applicability of indexed grammars to natural languages. In Uwe Reyle and Christian Rohrer, editors, *Natural Language Parsing and Linguistic Theories*. Reidel Dordrecht, 1988.

[Gis81] Jay L. Gischer. Shuffle languages, Petri nets, and context-sensitive grammars. *Communications of the ACM*, 24(9):597–605, 1981.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[Glu61] Victor Michailowitsch Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1–53, 1961.

[Göt08] Daniel Norbert Götzmann. Multiple context-free grammars. Technical report, Universität des Saarlandes, 2008.

[GS65] Seymour Ginsburg and Edwin H. Spanier. Mappings of languages by two-tape devices. *J. ACM*, 12:423–434, July 1965.

[Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer, 1992.

[HMU03] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Ed.)*. Pearson Education International, 2003.

[JLT75] Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. Tree adjunct grammars. *J. Comput. Syst. Sci.*, 10(1):136–163, 1975.

[Jos85] Aravind K. Joshi. Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural description? *Natural Language Processing — Theoretical, Computational and Psychological Perspective*, 1985.

[JS01] Joanna Jedrzejowicz and Andrzej Szepietowski. Shuffle languages are in P. *Theorical Computer Science*, 250(1-2):31–53, 2001.

[JSW90] Aravind K. Joshi, K. Vijay Shanker, and David J. Weir. The convergence of mildly context-sensitive grammar formalisms, 1990.

[Kle98] Philip N. Klein. Computing the edit-distance between unrooted ordered trees. In *In Proceedings of the 6th annual European Symposium on Algorithms (ESA*, pages 91–102. Springer-Verlag, 1998.

[KNSK92] Y. Kaji, R. Nakanisi, H. Seki, and T. Kasami. The universal recognition problem for multiple context-free grammars and for linear context-free rewriting systems. *IEICE Transactions on Information and Systems*, E75-D(1):78–88, 1992.

[Lev66] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[LW87] Klaus-Jörn Lange and Emo Welzl. String grammars with disconnecting or a basic root of the difficulty in graph grammar parsing. *Discrete Applied Mathematics*, 16:17–30, 1987.

[MRS98] Alexandru Mateescu, Grzegorz Rozenberg, and Arto Salomaa. Shuffle on trajectories: syntactic constraints. *Theoretical Computer Science*, 197(1-2):1–56, 1998.

[MS94] Alain J. Mayer and Larry J. Stockmeyer. Word problems – this time with interleaving. *Information and Computation*, 115:293–311, 1994.

[ORR78] William F. Ogden, William E. Riddle, and William C. Rounds. Complexity of expressions allowing concurrency. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 185–194, New York, NY, USA, 1978. ACM.

[Pol84] Carl Pollard. *Generalized phrase structure grammars, head grammars and natural language*. PhD thesis, Stanford University, 1984.

[Sel77] Stanley M. Selkow. The tree-to-tree editing problem. *Inf. Process. Lett.*, 6(6):184–186, 1977.

[Sha78] Alan C. Shaw. Software descriptions with flow expressions. *IEEE Trans. Softw. Eng.*, 4:242–254, May 1978.

[SMFK91] Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. On multiple context-free grammars. *Theor. Comput. Sci.*, 88(2):191–229, October 1991.

[Ste87] Mark Steedman. Combinatory Grammars and Parasitic Gaps. *Natural Language and Linguistic Theory*, 5:403–439, 1987.

[Tai79] Kuo-Chung Tai. The tree-to-tree correction problem. *J. ACM*, 26:422–433, July 1979.

[VdlC02] Éric Villemonte de la Clergerie. Parsing mildly context-sensitive languages with thread automata. In *Proceedings of the 19th international conference on Computational linguistics - Volume 1*, COLING '02, pages 1–7, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[Wei88] David J. Weir. *Characterizing mildly context-sensitive grammar formalisms*. Graduate School of Arts and Sciences, University of Pennsylvania, 1988.

[Wei92] David J. Weir. Linear context-free rewriting systems and deterministic tree-walking transducers. In *Proceedings of the 30th annual meeting on Association for Computational Linguistics*, ACL '92, pages 136–143, Stroudsburg, PA, USA, 1992. Association for Computational Linguistics.

[WH84] Manfred K. Warmuth and David Haussler. On the complexity of iterated shuffle. *J. Comput. Syst. Sci.*, 28(3):345–358, 1984.

[ZS89] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.

[ZSS92] Kaizhong Zhang, Rick Statman, and Dennis Shasha. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42(3):133 – 139, 1992.

Department of Computing Science
Umeå University, SE-901 87 Umeå, Sweden
www.cs.umu.se

ISBN 978-91-7601-047-1
ISSN 0348-0542
UMINF 14.13