Tree-Structured Problems and Parallel Computation

Dissertation

der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von

Dipl.-Inf. Michael Ludwig
aus Ludwigsburg

Tübingen
2018

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	4.10.2018
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Klaus-Jörn Lange
2. Berichterstatter:	Dr. Andreas Krebs

Zusammenfassung (Abstract in German)

Turing-Maschinen sind das klassische Beschreibungsmittel für Wortsprachen und werden daher auch benützt, um Komplexitätsklassen zu definieren. Dies geschieht zum Beispiel durch das Einschränken des Platz- oder Zeitaufwandes der Berechnung zur Lösung eines Problems. Für sehr niedrige Komplexität wie etwa sublineare Laufzeit, werden Schaltkreise verwendet. Schaltkreise können auf natürliche Art Komplexitäten wie etwa logarithmische Laufzeit modellieren. Ebenso können sie als eine Art paralleles Rechenmodell gesehen werden. Eine wichtige parallele Komplexitätsklasse ist $\mathbf{NC}^1$. Sie wird beschrieben durch Boolesche Schaltkreise logarithmischer Tiefe und beschränktem Eingangsgrad der Gatter.

Eine initiale Beobachtung, die die vorliegende Arbeit motiviert, ist, dass viele schwere Probleme in $\mathbf{NC}^1$ eine ähnliche Struktur haben und auf ähnliche Art und Weise gelöst werden. Das Auswertungsproblem für Boolesche Formeln ist eines der repräsentativsten Probleme aus dieser Klasse: Gegeben ist hier eine aussagenlogische Formel samt Belegung für die Variablen; gefragt ist, ob sie zu wahr oder zu falsch auswertet. Dieses Problem wird in $\mathbf{NC}^1$ gelöst durch den Algorithmus von Buss. Auf ähnliche Art können arithmetische Formeln in $\#\mathbf{NC}^1$ ausgewertet oder das Wortproblem für Visibly-Pushdown-Sprachen gelöst werden. Zu besagter Klasse an Problemen gehört auch Courcelles Theorem, welches Berechnungen in Baumautomaten involviert. Zu bemerken ist, dass alle angesprochenen Probleme gemeinsam haben, dass sie aus Instanzen bestehen, die *baumartig* sind. Formeln sind Bäume, Visibly-Pushdown-Sprachen enthalten als Wörter kodierte Bäume und Courcelles Theorem betrachtet Graphen mit beschränkter Baumweite, d.h. Graphen, die sich als Baum darstellen lassen. Insbesondere Letzteres ist ein Schema, das häufiger auftritt. Zum Beispiel gibt es **NP**-vollständige Graphprobleme wie das Finden von Hamilton-Kreisen, welches unter beschränkter Baumweite in **P** fällt. Neuere Analysen konnten diese Schranke weiter zu $\mathbf{SAC}^1$ verbessern, was eine parallele Komplexitätsklasse ist.

Die angesprochenen Probleme kommen aus unterschiedlichen Bereichen und haben individuelle Lösungen. Hauptthese dieser Arbeit ist, dass sich diese Vielfalt vereinheitlichen lässt. Es wird ein generisches Lösungskonzept vorgestellt, welches darauf beruht, dass sich die Probleme auf ein Termevaluierungsproblem reduzieren lassen. Kernstück ist daher ein Termevaluierungsalgorithmus, der unabhängig von der Algebra, über welche der Term evaluiert werden soll, ist.

Resultat ist, dass eine Vielzahl, darunter die oben angesprochenen Probleme, sich auf analoge Art lösen lassen, und dass sich ebenso leicht neue Resultate zeigen lassen. Diese Menge an Resultaten hätte sich ohne den vereinheitlichten Lösungsansatz nicht innerhalb des Rahmens einer Arbeit wie der vorliegenden zeigen lassen.

Der entwickelte Lösungsansatz führt stets zu Schaltkreisfamilien polylogarithmischer Tiefe. Es wird jedoch auch die Frage behandelt, wie mächtig Schaltkreisfamilien konstanter Tiefe noch bezüglich Termevaluierung sind. Die Klasse $\mathbf{AC}^0$ ist hierfür ein natürlicher Kandidat; sie entspricht der Menge der Sprachen, die durch Logik erster Ordung beschreibbar sind. Um dieses Problem anzugehen, wird zunächst das Termevaluierungsproblem über endlichen Algebren betrachtet. Dieses wiederum lässt sich in das Wortproblem von Visibly-Pushdown-Sprachen einbetten. Daher handelt dieser Teil der Arbeit vornehmlich von der Beschreibbarkeit von Visibly-Pushdown-Sprachen in Logik erster Ordnung. Hierbei treten ungelöste Probleme zu Tage, welche ein Indiz dafür sind, wie schlecht die Komplexität konstanter Tiefe bisher noch verstanden ist, und das, trotz des Resultats von Furst, Saxe und Sipser, bzw. Håstads.

Die bis jetzt beschrieben Inhalte sind Teil einer kontinuierlichen Entwicklung. Es gibt jedoch ein Thema in dieser Arbeit, das orthogonal dazu ist: Automaten und im speziellen Cost-Register-Automaten. Zum einen sind, wie oben angedeutet, Automaten Beispiele für Anwendungen des hier entwickelten generischen Lösungsansatzes. Zum anderen können sie selbst zur Beschreibung von Termevaluierungsproblemen dienen; so können Visibly-Pushdown-Automaten Termevaluierung über endlichen Algebren ausführen. Um über endliche Algebren hinauszugehen, benötigen die Automaten mehr Speicher. Visibly-Pushdown-Automaten haben einen Keller, der genau dafür geeignet ist, die Baumstruktur einer Eingabeformel zu verifizieren. Für nichtendliche Algebren eignet sich ein Modell, welches hier vorgestellt werden soll. Es kombiniert Visibly-Pushdown-Automaten mit Cost-Register-Automaten. Ein Cost-Register-Automat ist ein endlicher Automat, welcher mit zusätzlichen Registern ausgestattet ist. Die Register können Werte einer Algebra speichern und werden in jedem Schritt in Abhängigkeit des Eingabezeichens und des Zustandes aktualisiert. Dieser Einwegdatenfluss von Zuständen zu Registern sorgt dafür, dass dieses Modell nicht nur entscheidbar bleibt, sondern, in Abhängigkeit der Algebra, auch niedrige Komplexität hat. Das neue Modell der Cost-Register-Visibly-Pushdown-Automaten kann nun Terme evaluieren. Es werden grundlegende Eigenschaften gezeigt, einschließlich Komplexitätsaussagen.

Acknowledgements

First and foremost, my thank goes out to my advisors Klaus-Jörn Lange and Andreas Krebs. I greatly appreciate the freedom and trust I enjoyed. I also want to thank Nutan Limaye who, beside my advisors, is my main collaborator, and helped me to shape some of the main ideas in this work. Also, she has been a wonderful host on my three research visits to the Indian Institute of Technology in Mumbai. I thank Volker Diekert who guided my first steps in theoretical computer science.

I thank Hanspeter Hägele for his intensive investment in proofreading this thesis to fix linguistic shortcomings. Two other linguistic talents who helped me with some sections are Anne Bernhardt and Maria Panter.

Moreover, I thank all the nice people I got to know over the last years while I was teaching and doing research at the University of Tübingen; especially the (former) members of the TI department should be mentioned (ordered alphabetically): Michaël Cadilhac, Silke Czarnetzki, Olga Dorzweiler, Demen Güler, Renate Hallmayer, Klaus Reinhardt, Sebastian Schöner, Ingo Skupin, Thomas Stüber, and Petra Wolf.

Lastly, I also thank my parents, Manfred and Waltraud Ludwig, and all my friends.

Contents

Chapter 1

Introduction

In the first section, which for the most part should also be accessible to the non-theoretician, we give a rather informal overview of this thesis. After, we detail how this work is structured and motivated.

1.1 Outline

Computer science deals with the systematics behind the representation, storage, and manipulation of information in theory and practice. The field emerged from two different directions: On the one hand, we have engineering, which deals with building hard- and software with an aim for being used in production. Scientific knowledge is predominantly obtained here by empirical methods and experimentation. On the other hand, there is the direction that originated in mathematics where results are gathered by mathematical proofs. While both directions are complementary and as such have different scopes, *theoretical computer science* (TCS), the field this work is located in, is one of the descendants of only the latter. The results achieved in TCS are usually rather weak and very hard to obtain, but at the same time they, like any other mathematical result, are timeless and have universal validity.

Algorithms is a subfield of TCS that represents a close link to practice. In it, one tries to obtain formal procedures to solve computational problems within some computational model. For this work it is important to note that this not only includes classical imperative programming, but all ways to formally describe a computation procedure. The power and limitations of computational models themselves are being analyzed in *complexity theory*. The powers of computational models are organized in complexity classes. Some of the best known examples are **P** and **NP**.

The class **P** contains all problems that are solvable by a deterministic computer in polynomial time, whereas the superclass **NP** also contains problems that are solvable in polynomial time using non-determinism. The problems in **P** are often considered efficiently solvable, while **NP** is regarded as a class that contains problems that cannot be efficiently solved in general; we only know deterministic algorithms that are exponential in their runtime. Although, it should be mentioned that in practice the problems in **NP** can often be solved reasonably well by using approximations or heuristics, which work on most inputs. As of now, the status of the $\mathbf{P} \overset{?}{=} \mathbf{NP}$ problem is unknown.

Within the class **P** there exists a subclass called **NC** for which it is unknown whether it is a proper subset of **P**, or not. Intuitively speaking, this class contains all problems that can be efficiently parallelized. Such problems need polylogarithmic time to solve, given sufficiently many processors. In this work, we will consider such parallelizable problems. More precisely, most of this work will, in one way or another, revolve around the class $\mathbf{NC}^1$ which, in contrast to **NC**, only allows logarithmic time.

The parallel complexity classes are actually circuit complexity classes. A Boolean circuit consists of Boolean gates and wires between the gates and works in the expected way. The interpretations of parallel computers and circuits are equivalent. Here, the depth of circuits corresponds to the time a parallel algorithm needs. In $\mathbf{NC}^1$ there are all problems that can be solved using circuits of logarithmic depth, or, if seen from the perspective of parallel computation, all problems that can be solved in logarithmic time.

The class $\mathbf{NC}^1$ is of interest since many natural problems are located here and many of them are $\mathbf{NC}^1$-hard, which means that they are indeed at the tight upper end of this complexity class. The first key observation constituting the research done in this thesis is the following:

Many problems in $\mathbf{NC}^1$ have an underlying tree-like structure.

Let us make this rather heuristic statement more concrete. Whenever we say a problem has an underlying tree-like structure we mean that the language or function in question takes inputs that are a tree or forest, or at least are highly hierarchical. There are problems that naturally have a tree structure like solving the word problem for tree languages. But also, if we consider problems for general graphs and then restrict them to trees or tree-like graphs, we often arrive at the $\mathbf{NC}^1$-bound or something similar. Examples include Courcelle's Theorem and **NP**-complete graph problems restricted to such classes of graphs. Here, tree-like graphs means graphs of bounded tree- or clique-width.

Now, the second key observation is that many $\mathbf{NC}^1$ problems not only have an underlying tree-like structure, but also:

The $\mathbf{NC}^1$-algorithms[1] to solve those problems are often very similar conceptionally and are based on the same ideas and principles.

So, this set the goal to find a unified approach that generalizes as many results as possible.

When we say a problem has a tree-like structure, by that we mean that the inputs we consider represent trees in some way. These trees have some semantic on which the computation that is performed depends. Terms and algebras capture these semantics. Terms are trees where each node is assigned some operation of the algebra, so there is a very close relationship. This is the third key insight:

Most problems that have an underlying tree-like structure can be reduced to a term evaluation problem.

This means that if we have the means to evaluate terms over some algebra, we can also find a solution for all problems that can be reduced to this term evaluation. Hence, in this work we develop a term evaluation framework, which is independent of the algebra, and we show how to use it for deriving upper bounds: We show how to use it in general and also give examples in terms of specific problems.

Besides showing upper bounds for general term evaluation, we will also investigate the evaluation capabilities of complexity classes weaker than $\mathbf{NC}^1$. In particular we look into the question of *how much* evaluation is possible in constant depth circuits. Although this question is natural in itself, we can give another motivating perspective:

Many $\mathbf{NC}^1$-problems we consider originate in more complex problems; **NP**-hard problems, for example. These problems became parallelizable by enforcing tree-like inputs. We can now ask for other classes how we have to restrict the inputs in order to reduce the complexity to the language in question. Since enforcing tree-like inputs corresponds to logarithmic depth, we continue by considering constant depth, i.e. $\mathbf{AC}^0$. Since $\mathbf{AC}^0$ is a subset of $\mathbf{NC}^1$ we need a stronger restriction.

The property we use has many characterizations and expresses that the input not only is a tree, but that these trees also have to have a limited branching complexity. One of these characterizations lies in assuming a bounded Horton-Strahler number. If one considers the spectrum between complete binary trees and degenerated trees, which are basically lists, a bounded Horton-Strahler number forces the trees to be

[1]As we already mentioned, algorithms can be implemented in various computational models. Here, $\mathbf{NC}^1$-*algorithm* means a circuit family or, equivalently, an alternating Turing machine.

more similar to a list. Summarizing, while a problem in general might be in **NP**, the problem bounded to tree-like inputs could be in logarithmic depth and the problem bounded to tree-like inputs with limited branching complexity in constant depth.

A final aspect are automata. On the one hand they serve as application examples: Using our term evaluation algorithm we can solve the word problem for certain kinds of automata in $\mathbf{NC}^1$. On the other hand the automata we consider can capture evaluation problems themselves and are closely related to algebra, so they serve as a tool, especially for the $\mathbf{AC}^0$ bounds.

In summary, this work consists of the following two parts. Both partly contain original content as well as pre-known content. The preliminaries to understand the main benefits of this work are mostly taken care of in Part I, whereas Part II contains the bulk of the main results and should be considered the more important part.

- Part I: Modeling
  - A framework to formulate the term evaluation problems in, which mostly consists of algebra.
  - The machinery in which we implement the algorithms, which consists of a generalized version of circuits and logic.
  - Different automaton models.
- Part II: Evaluation
  - The general algorithm for term evaluation.
  - Applications to tree-structured problems.
  - Algorithms for the evaluation of terms that have a limited branching complexity.

In the next sections we give a more in-depth overview of the contents of this work.

1.2 Background

In this section we observe how the results in different areas were obtained, how they are related, and how they are relevant. There are four major blocks we consider: Evaluation problems, automata related problems, problems parameterized by tree-width, and low complexity. All four represent a distinct line of research.

The way this section is arranged aligns with the order of the chapters in the second part.

Evaluation Problems

The satisfiability problem (SAT) can be considered the initial **NP**-complete problem [Coo71]. For this problem we are given a propositional formula containing variables. Then however, the question arises whether there exists a valuation for the variables such that the formula evaluates to true. To solve the problem, the **NP**-algorithm first guesses the valuation of the variables using non-determinism and then checks whether the formula evaluates to true. We are interested in the last step, which is called the Boolean formula value problem (BFVP). Of course, this check can be done in **NP**, but could a lower complexity be sufficient? This question has received a lot of research, which continued to lower the upper bound. The endpoint of this trend was [Bus87] in which Buss showed that the BFVP is in $\mathbf{NC}^1$. The proof for this result involves sophisticated game arguments and reasoning over the input formula, although in a follow-up paper, an alternative simplified proof was presented [Bus93].

At this point it is important to note that formulas are basically labeled ranked trees. A tree can be represented as a word that is a parenthesized expression. Checking whether an input formula codes a valid tree at all is a problem in $\mathbf{TC}^0$. This problem can be easily reduced to the word problem of the Dyck language.

As a result of the progression of the upper bound research for the BFVP, the complexity of evaluating arithmetic formulas has also been considered. Evaluating terms over the natural numbers together with addition and multiplication is a problem in $\#\mathbf{NC}^1$ [BCGR92]. The proof of this built upon [Bus87].

The main idea for evaluating Boolean formulas is to employ a divide and conquer algorithm. For example, to evaluate a given input formula, we would evaluate certain subintervals of the input parallely and then use those recursive results to compute the overall output. As a schematic example, consider Figure 1.1. Here, we have a term that is a balanced tree. One could recursively evaluate the left and right subtree of the root and then obtain the overall result, but in general, we do not know in advance how the input will be structured. It could also be a degenerated tree like in Figure 1.2. In this case if we split the formula in half, the left part would be a subformula, but the right part would not be a proper formula any more. The main idea in [Bus87] is to evaluate in parallel the left part and the right part as well, but the right part has to be evaluated twice: Once with the gap filled by true and once filled by false; see Figure 1.3.

For the case of arithmetic formulas this approach does not work any more since we would need to evaluate the right part for the infinite number of possible outcomes of the left part. The idea to solve this problem is presented in [BCGR92]. In it the authors observe that the right part is a term with a hole and this as a whole evaluates to a function $\mathbb{N} \to \mathbb{N}$ that is of the very simple form $x \mapsto ax + b$. It is

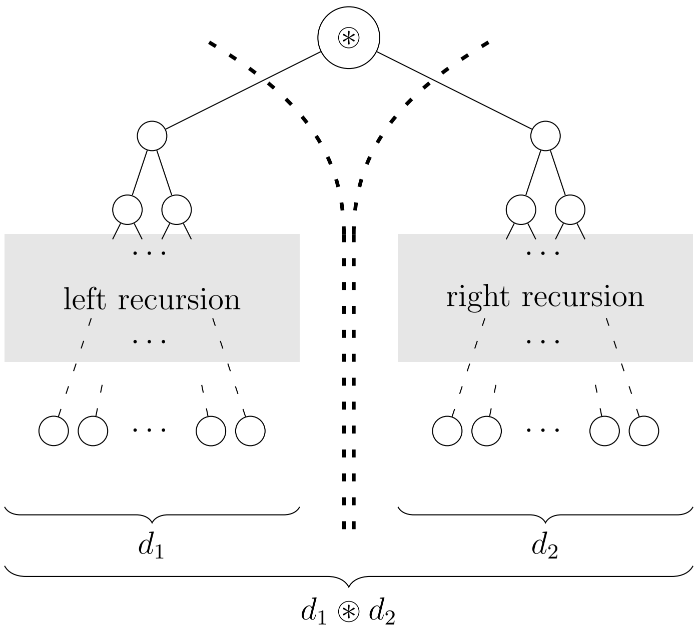


Figure 1.1: Terms are trees; the figure indicates a balanced tree, which resembles how the term is composed. For a recursive evaluation approach it is a sensible initial idea to make a split in the middle to recursively evaluate the left and the right subtrees of the root generating the results d_1 and d_2, and then using these results to obtain the overall result as $d_1 \circledast d_2$.

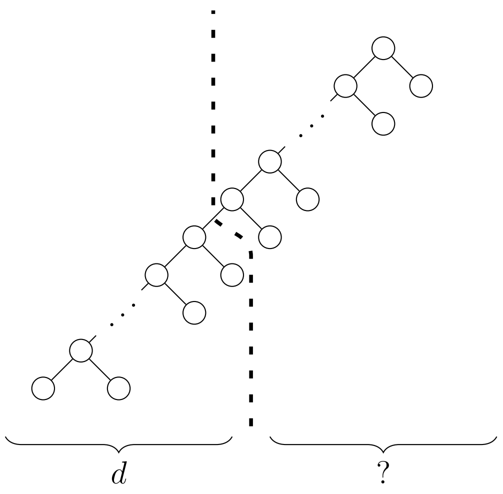


Figure 1.2: In contrast to the situation seen in figure 1.1, the approach of splitting the term in the middle is not as straight forward this time where the tree is degenerated.

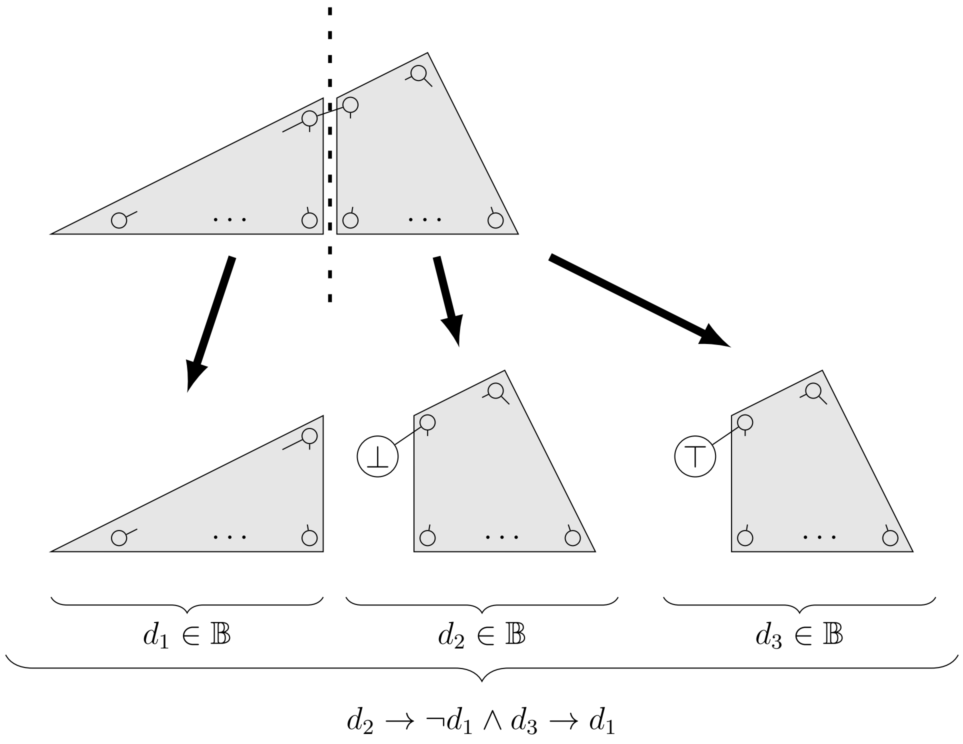


Figure 1.3: When evaluating a Boolean formula recursively one can split the term and evaluate the left part. The recursion over the right part is done twice; once for $\perp$ and once for $\top$ as substitutes for the left part. The three results are then being combined.

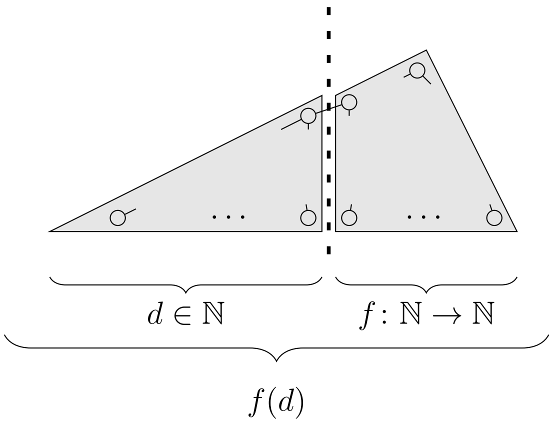


Figure 1.4: Arithmetic formulas can be recursively evaluated as shown in the figure. The key here is that f is of the simple form. This form, as it will turn out later, is $x \mapsto ax + b$. Thus, the evaluation of the right part will in fact provide the values a and b. Then $f(d) = ad + b$. This idea we will, as one of our main contributions, generalize to arbitrary algebras.

shown that it is, in fact, possible to compute a and b. Afterwards, the evaluation of the left part is inserted for x and we get the overall result; see Figure 1.4.

A thorough understanding of this idea is important for the comprehension of the main result in which we will take this idea and develop it further, so it is not only applicable in specific situations like evaluating terms over the natural numbers.

Automata Related Problems

Automata theory is the second line of research we consider. Some results in this area are based on findings described previously. In particular, the complexity of the word problem for certain automata will be considered. The standard example to begin with are the regular languages, which are in $\mathbf{NC}^1$. The construction used to show this is based on the fact that regular languages are recognized by homomorphisms and finite monoids. Checking whether a word is a member of some regular language is equivalent to multiplying elements in a finite monoid. Associativity is the property which, in this case, makes the problem very easy since since the computation does not have to deal with hierarchical inputs.

A generalization of the regular word languages are the so-called visibly pushdown languages (VPLs) [AM04] introduced by Alur and Madhusudan using visibly pushdown automata (VPAs). They already existed under the name of input-driven pushdown languages [Meh80] as introduced by Mehlhorn. In [Dym88], Dymond built on the core algorithm for the BFVP and adapted it to visibly pushdown languages.

This was a first indication that the result of Buss has potential that goes well beyond evaluating formulas itself.

Visibly pushdown languages are equivalent to regular nested word languages [AM09] which are languages over words that are equipped with a nesting relation in addition to the position order relation. Also, regular forest languages are isomorphic to visibly pushdown languages[2]. From that perspective it is easy to see that evaluating terms over any finite algebra is nothing else than computations of ranked tree automata. Hence, the evaluation of terms over finite algebras can be reduced to visibly pushdown automata computations - a fact, which will be used later.

All three automata types we mentioned have a word problem that is in $\mathbf{NC}^1$. It is interesting to compare this fact to the situation for regular languages. Basically, this tells us that associativity is not needed to keep the complexity in $\mathbf{NC}^1$. Or, to put it differently: The input may be truly hierarchical as opposed to the case of regular languages whose elements represent lists, i.e. degenerated trees.

The result of Dymond [Dym88] for the Boolean case has been extended even further: The complexity of counting accepting computations in non-deterministic visibly pushdown automata is in $\#\mathbf{NC}^1$ [KLM12] which comes as no surprise. The proof built on Dymond's proof strategy, but it became increasingly complicated.

Generalizing finite automata to tree inputs is one way to go beyond classical word language setting. Another way to go beyond lies in going over to functions. We can associate languages with their characteristic functions that map to $\{0,1\}$. One can now come up with automata models that represent more complex functions, i.e. functions that map to larger sets, like $\mathbb{N}$. Relevant for this work are the following three variants:

- *Counting.* If we take a non-deterministic automaton, we can assign each input word the corresponding number of how many accepting computations there are.

- *Weighted automata.* These are based on a semiring $(R; \oplus, \otimes)$ and a non-deterministic automata. Each transition is assigned some weight. All weights along a run are being aggregated by $\oplus$. Then the obtained values for all runs are being aggregated by $\otimes$, which then becomes the output.

- *Cost register automata.* This type of automaton is based on deterministic finite automata and is equipped with registers over some algebra. The registers can be updated in each step according to the state. The output is the final value computed.

[2]That is, if we neglect internal letters.

Especially the last model is very interesting as it is powerful but still tame enough to be analyzed.

Problems Parameterized by Tree-width

Results about formula evaluation and, continued by, complexity results about automata are one line of research that can be traced back to the algorithm of Buss. However, there exist many other results that show upper bounds in terms of parallel complexity for problems that, in some sense, are also tree-like.

One major class of this kind of problems consists of graph problems that drop in complexity if the input graph is known to be tree-like. In our case tree-like can mean either that the tree-width or the clique-width of the input graph is bounded. In both cases the input graph can be represented as a term whose evaluation is the original graph. Many problems become easier if we assume a bounded width. The prime example is the Theorem of Courcelle [Cou90]. It states that checking whether an input graph of some fixed tree-width is a model for a fixed MSO formula, can be done in linear time. The result consists of two steps: Decomposing the input graph, which yields a term, and then checking whether the term satisfies the formula. For the second step, which is the one we are interested in, the formula is transformed into a tree automaton. The first step also received research [EJT10] which led to the complexity of finding the decomposition being reduced to logspace. The second step is of complexity $\mathbf{NC}^1$ which is what we would expect [EJT12].

Another example are **NP**-complete graph problems. One of which is the problem of finding Hamiltonian cycles in graphs. In [Wan94] Wanke showed that this problem is in **P** if bounded clique-width is assumed. It is also possible for this problem to be brought down to $\mathbf{SAC}^1$ [BDG15]. Another graph problem that behaves this way is the one concerned with finding maximal cuts.

Both for Courcelle's Theorem and the mentioned graph problems one can also consider counting versions leading to the respecting counting complexities.

A third problem of this kind was recently investigated in [JS14] whose results state that a Boolean circuit family of polynomial size can be balanced if the graphs of the circuits have a bounded tree-width. This means that if a problem has a polynomial size circuit family of bounded tree-width, the problem is already in $\mathbf{NC}^1$.

As one can see, there are ample problems whose results do not rely on the Buss algorithm, but rather go their own ways.

Low Complexity

We already mentioned the regular languages, which are contained in the class $\mathbf{NC}^1$. A natural follow-up question to ask is which regular languages are contained in certain subclasses of $\mathbf{NC}^1$. One instance would be asking for the regular languages that are in $\mathbf{AC}^0$, that is constant depth circuits, or, equivalently that are definable in first-order logic using arbitrary numerical predicates. The motivation of this problem actually is similar to a case we already considered: We have asked how we have to restrict input graphs for an MSO formula such that the model checking problem can be solved parallely. For an $\mathbf{NC}^1$ problem the question we can ask is how we have to restrict the problem further in order to get to constant depth. In the case of regular languages this has been carried out already and as a result there is a decidable algebraic characterization of the regular languages that are first-order definable [BCST92]. Here, we are in the situation of even being able to decide when a regular language is in $\mathbf{AC}^0$ or not. This is due to the fact that the parity language is not in $\mathbf{AC}^0$ [FSS84, Hås87] which is one of the strongest complexity theoretic results we have. Beyond regular word languages, little is known in this direction.

1.3 Main Contributions

The historic background we have laid out in the previous section served to motivate the topic of this thesis. Notice that we listed several problems and their complexities that get an input that is in some way tree-shaped; either by actually being a tree or by graphs of bounded width, or by formulas. All those complexity results concern upper bounds in terms of parallel complexity classes and their proof strategies also follow the same ideas. Here, the wheel has been invented several times.

Our goal is to present a framework that unifies all mentioned results. Our framework centers around term evaluation over arbitrary algebras. This is because of the observation that all mentioned problems are reducible to a term evaluation problem and still we are lacking such a general term evaluation algorithm. Accordingly the first main aspect of this work is:

Term Evaluation Over Arbitrary Algebras

We already outlined the evaluation algorithms for Boolean and arithmetic formulas. Now, we take this further to arbitrary algebras. It needs to be stressed that we do not assume any property for the algebra. For example, neither associativity, nor distributivity must hold. The algebras may have any number of operations of any finite arity. Also, the finiteness of the domain is not required. As a consequence the complexity result is formulated in dependence of the algebra: The complexity of evaluating a term over an arbitrary algebra $\mathcal{A}$ is $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$, where $\mathcal{F}(\mathcal{A})$ is an algebra based on $\mathcal{A}$ and $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$ is represented by logarithmically deep circuits that, in a nutshell, have oracle gates that are operations of $\mathcal{F}(\mathcal{A})$.

The key to obtain this result lies in generalizing the idea of arithmetic formula evaluation as outlined before. Here, the formula is split and the left and the right part are evaluated in parallel, however, the right part does not evaluate to a number but to a function. Now, in the case of arbitrary algebras $\mathcal{F}(\mathcal{A})$ captures precisely the functions we get when making such a split; it contains, in addition to the domain and operations of $\mathcal{A}$, all the functions that might occur.

The deeper details of the algorithm we will present are a bit technical, but the overall structure is easy to grasp and we claim that our direct construction is more accessible than the rather indirect approaches of Buss et al. in [Bus87] and [BCGR92].

After we have established this main tool, we will show how to apply it in order to obtain upper bounds. We also demonstrate that this has several concrete applications:

Application of the Term Evaluation Algorithm

We want to solve problems, which means showing upper bounds. For us the main premise is that there are many relevant problems that can be reduced to a term evaluation problem. Hence, we will present a template to obtain actual upper bounds. Note that the complexity $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$ we obtained in the main algorithm is somewhat artificial and it is unclear how it relates to the usual complexity classes. Our template consists of first reducing the problem to an evaluation problem, placing it in $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$, and then relating $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$ to known classes.

We then will proceed to show several upper bounds using the template and algorithm. In the previous section we introduced the problems already.

- *Evaluating Boolean formulas.* This application is trivial for us. We claim that we can reproduce the original results of Buss [Bus87]. We also obtain the folklore result that evaluating terms over finite algebras is in $\mathbf{NC}^1$.

- *Evaluating arithmetic formulas.* This easy application reproduces the result of Buss et al. [BCGR92]. We also derive that, in general, evaluating terms over a distributive algebra $\mathcal{A}$ is in $\mathcal{A}$-$\mathbf{NC}^1$.

- *Automata.* We consider a variety of different problems for different automata models. The are either based on words or trees whereas the word automata comes in the shape of visibly pushdown automata. There is also the distinction between Boolean automata that compute a single output bit and more general models such as weighted and cost register ones.

Depending on the model, the different problems we look at include the word problem, the uniform word problem in which the automaton is part of the input, and counting.

- *Circuits of bounded tree-width.* We show that Boolean circuit families of polynomial size and bounded tree-width collapse from $\mathbf{P}$ to $\mathbf{NC}^1$.
- *Courcelle's Theorem.* Here, we consider the part of the problem where the input graph is already decomposed and obtain the $\mathbf{NC}^1$ bound. We also reproduce a counting variant where the MSO formula has a free second-order variable and we count how many valuations there are such that the graph satisfies the formula.
- $\mathbf{NP}$*-complete problems parameterized by NLC-width.* Wanke showed that finding Hamiltonian cycles and maximal cuts in graphs is in $\mathbf{P}$ if bounded width is assumed [Wan94]. In [BDG15] the bound for the Hamiltonian cycle problem was reduced to $\mathbf{SAC}^1$ and also counting the number of cycles was considered resulting in the complexity $\#\mathbf{SAC}^1$. We reproduce this result and also show the analogue result for the maximal cut problem.

One big part in the applications have been automata. They actually play such a significant role that they deserve to be called a major aspect themselves. Besides being important examples for applications, they can also to be considered objects that evaluate terms. Especially the newly introduced cost register VPAs implement this. On the other hand, the Boolean VPAs give rise to an algebraic treatment of tree languages and hence evaluation, especially in the low complexity case.

Automata for Expressing Languages and Functions and Algebraic Characterizations

For one, we survey known automata models. Besides those that are based on ordinary finite word automata, all transcend them by considering tree-like inputs. This includes nested word automata, visibly pushdown automata and actual tree automata. There is a great variety of different models and many interconnections.

Besides the Boolean models, we also examine counting, weighted models and cost register models. The latter we merged with visibly pushdown automata to get a model that can handle tree-shaped inputs and perform arithmetic operations on it.

Aside from the treatment of automata themselves we also consider the closely related subject of algebra. Algebra plays two roles in this work. First, evaluation problems are based on algebras. Second, we use algebras to capture computations, which is done by syntactic algebras. We develop an algebraic framework to capture visibly pushdown languages and other tree-structured language classes. For forest languages there already exist the forest algebras by Bojańczyk and Walukiewicz [BW08] which we embed into our algebraic framework. In particular we present *extend algebras* which are a meaningful alternative to forest algebras.

We also present a characterization of functions implemented by finite cost register automata in terms of a wreath product. This product underpins the fact that the states of such an automaton direct the register updates. This one-way flow of information is what makes the model tame and is exactly what the wreath product expresses.

Finally, we consider evaluation in low complexity. This leads to a combination of all previous parts. Instead of asking directly for evaluation in low complexity, we ask which VPLs are in constant depth. One can also look at this from the perspective of, for example, graph problems like Courcelle's Theorem. The property that brings down the complexity to logarithmic depth is bounded clique-width. Now, we ask: What property do we have to postulate to bring down the complexity to constant depth.

Evaluation in Constant Depth

A pattern we observed is that problems drop to logarithmic depth if tree-shaped inputs are assumed. Now, we ask the question how problems have to be restricted in order to let the complexity drop to constant depth. In general, we are interested in evaluation problems. Since we cannot provide an exhaustive result as in Chapter 9, we only consider finite algebras in this part. Evaluation problems over finite algebras can be embedded into VPLs. Also, the constant depth class we are interested in is $\mathbf{AC}^0$ which equals first-order logic over arbitrary numerical predicates. What we do, instead of asking what we can evaluate within $\mathbf{AC}^0$, is posing the question which VPLs are first-order definable. This has the benefit of being linked to another successful line of research for which we hope that many results carry over. VPLs generalize regular languages for which a characterization of the ones that are first-order definable already exists: Regular languages that are first-order definable are precisely those that have a quasiaperiodic syntactic homomorphism. We try to generalize this to VPLs.

We are able to split the process of solving the word problem of VPLs into two parts. The first part addresses the complexity of the tree that the words in the language represent. Here, a tree of low complexity could be one that degenerated into a list, whereas a balanced binary tree is very complex. We capture this complexity formally and connect this notion to already known tree properties. Besides this tree property, the second part captures the analogue to quasiaperiodicity in the case of regular word languages. Unfortunately we do not get a tight result. The second part is tame, but it is very hard to determine which tree shapes can be analyzed by first-order formulas. This is embodied by an open question we pose that can be stated very succinctly: Is the language generated by the grammar rules $S \to aSbc|acSb|\epsilon$ first-order definable? We conjecture that it is not and that proving it could yield deep insights into $\mathbf{AC}^0$ since up to now the only tool we have to show that a language is not in $\mathbf{AC}^0$ is the proof for the parity language [FSS84, Hås87].

Up until now we only considered the question of which languages are expressible in first-order logic with arbitrary numerical predicates. It will turn out that the major challenge is to define a matching predicate within the logic. It is natural to ask now what happens if we artificially add this predicate to the logic. Hence, we investigate more settings than the one only using arbitrary numerical predicates.

1.4 About This Thesis

Structure

This thesis is split into a *Part I* and a *Part II*. The first part is called *Modeling* and deals with representing languages, functions, and structures, whereas the second part, *Evaluation Complexity*, uses the objects defined in the first part to formulate the main complexity results.

Thereby, the **first part** contains all the preliminaries to understand the main results, but it goes beyond just providing basic definitions. In several places existing notions have been extended significantly. This is especially true for chapters on algebra and automata but also for circuits. Moreover, results on these new objects are presented.

The **second part** has three chapters, which represent the main contribution chapters. Chapter 9 contains the general term evaluation algorithm and Chapter 10 the application framework and many concrete applications. Chapter 11 then considers low complexity evaluation. Note that this covers three of the four main items mentioned in the previous section, whereas the item addressing automata lies orthoganal to the other topics and appears throughout the thesis: Chapters 6, 7, and 11, as well as Section 10.3, represent this aspect in particular.

Figure 1.5 shows a summary over the chapters in this thesis.

We already discussed the structure of this thesis with respect to its scientific content. It is divided into two parts, each consisting of a number of chapters. Importantly,

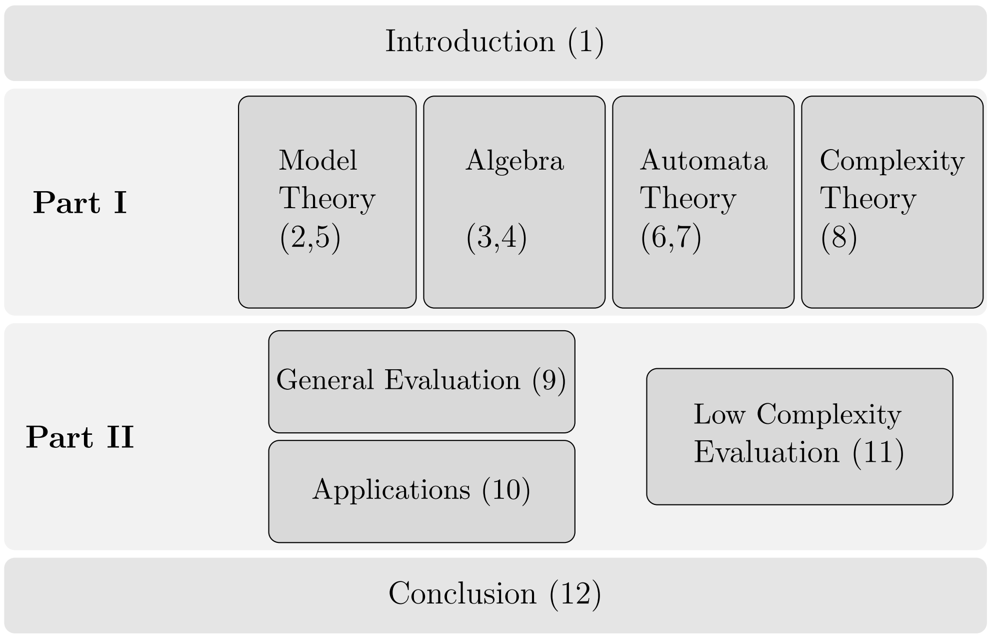


Figure 1.5: The thesis consists of two parts whereas the individual topics of Part II depend on the whole of Part I. Part I covers four research areas, which are represented by seven chapters.

each chapter has a conclusion section. The conclusion sections all follow the same format consisting of:

- Summary
- Contributions
- Sources and Related Work
- Further Research

We do this because each chapter represents a part of this work in its own right. Hence, there is no chapter that only serves as providing preliminaries. Furthermore, much of what could be considered preliminaries is actually generalized from the standard. For example, instead of ordinary algebras, we consider many-sorted families of algebras. The consequence is that there is no obvious line between original and previously known content. The conclusion sections for each chapter tries to clarify what is new and what is already known. Here, *new* can mean that something was introduced in a paper, which I have co-authored or that it never appeared in any paper before.

Contributed Papers

Up to the date of this writing I have published the following papers[3]:

[KLL17a] **A Unified Method for Placing Problems in Polylogarithmic Depth.** *Andreas Krebs, Nutan Limaye, Michael Ludwig*, FSTTCS 2017

[KLL17b] **A Unified Method for Placing Problems in Polylogarithmic Depth.** *Andreas Krebs, Nutan Limaye, Michael Ludwig.* ECCC 2017

[DFKL16] **Positive and negative proofs for circuits and branching programs.** *Olga Dorzweiler, Thomas Flamm, Andreas Krebs, Michael Ludwig.* TCS 2017

[KLL16] **Cost Register Automata for Nested Words.** *Andreas Krebs, Nutan Limaye, Michael Ludwig.* COCOON 2016

[KLL15a] **On Distinguishing $\mathbf{NC}^1$ and NL.** *Andreas Krebs, Klaus-Jörn Lange, Michael Ludwig.* DLT 2015

[CKLP15] **A Circuit Complexity Approach to Transductions.** *Michaël Cadilhac, Andreas Krebs, Michael Ludwig, Charles Paperman.* MFCS 2015

[HKLL15] **Visibly Counter Languages and the Structure of $\mathbf{NC}^1$.** *Michael Hahn, Andreas Krebs, Klaus-Jörn Lange, Michael Ludwig.* MFCS 2015

[KLL15b] **Visibly Counter Languages and Constant Depth Circuits.** *Andreas Krebs, Klaus-Jörn Lange, Michael Ludwig.* STACS 2015

[KLL14] **Visibly Counter Languages and Constant Depth Circuits.** *Andreas Krebs, Klaus-Jörn Lange, Michael Ludwig.* ECCC 2014

[DFKL14] **Positive and Negative Proofs for Circuits and Branching Programs.** *Olga Dorzweiler, Thomas Flamm, Andreas Krebs, Michael Ludwig.* DCFS 2014

This thesis contains contents of these papers to a varying degree. The papers [DFKL16, CKLP15, HKLL15, DFKL14] do not appear at all. The paper [KLL15b, KLL14] is the origin of Chapter 11, however, the research on that topic has advanced to a point that the result of the paper is a mere corollary of what is presented here, which is covered in Section 11.3. The papers [KLL15a, HKLL15] build on [KLL15b, KLL14], however, [HKLL15] does not get picked up here and [KLL15a] has one result that is the topic of Section 11.7.2. The paper [KLL16] was a prequel of [KLL17a, KLL17b]; the complete paper got incorporated. It appears in Chapter 7

[3]Papers are ordered chronologically and authors alphabetically.

and Section 10.3. The paper [KLL17a, KLL17b] then generalized the result greatly to the large set of applications already mentioned. This paper is the main foundation of this thesis and contents of it appear throughout it. In particular Chapter 9 originates in it: To a large extend it is adapted more or less word by word. To a lesser extend this also holds for parts of Chapter 10. Contrary to the case of the other papers that were incorporated in this work, it would have been awkward to rewrite Chapter 9 for the sole purpose of not repeating the exact same words.

It is worth pointing out how the thread throughout this thesis came together. Chapter 11 has its roots in [KLL15b, KLL14]. This was an early paper going into this research topic and was actually not related to evaluation in any way; it was purely concerned with the problem of first-order definability of visibly pushdown models. Later, seemingly unrelated papers [KLL16] and [KLL17a, KLL17b] initiated a new direction. It emerged afterwards that [KLL15b, KLL14] can be considered a paper about low complexity evaluation over finite algebras. This is the interpretation used here.

Part I

Modeling

Chapter 2

Structures

Before proceeding to the actual content of this chapter, some mathematical notation that holds for throughout the thesis has to be fixed.

2.1 Basic Notation

Although the reader is assumed to be familiar with set theory, we fix some notation here. Let X be a set, then $|X|$ denotes the *cardinality* of X, that is the number of elements in X.

The *powerset* of X is the set of all subsets of X and is denoted by 2^X or $\mathcal{P}(X)$. By $X \times Y$ we denote the *Cartesian product*. It contains pairs (x, y), where $x \in X$ and $y \in Y$. Most of the time, but not always, we use the Cartesian product in its associative version, i.e. for another set Z we get $(X \times Y) \times Z = X \times (Y \times Z) = X \times Y \times Z$. Therefore, for $((x, y), z) \in (X \times Y) \times Z$ we may say $((x, y), z) = (x, y, z)$. We call this *flattening*. It is clear from the context whether a Cartesian product is to be understood as flattened or not, i.e. as associative or not.

The set $\{0, 1, 2, 3, \ldots\}$ is called the *natural numbers* and denoted as $\mathbb{N}$. For $i \leq j$ we denote an interval $\{i, \ldots, j\}$ as $[i, j]$. Further, we define $[n]$ as $[1, n]$. The set of *integers* we denote as $\mathbb{Z}$. There is also the set $\mathbb{B} = \{\perp, \top\}$ where $\perp$ is the Boolean value for *false* and $\top$ the Boolean value for *true*. Depending on the context we may also use 0 and 1 for false and true. The symbols $\mathbb{Q}$ addresses the rational numbers.

For $n, m \in \mathbb{N}$ the set X^n is the associative Cartesian product $X \times \ldots \times X$ consisting of n-tuples, which we also call *vectors* or *words* - depending on the context. By X^* we denote the set $\bigcup_{n \in \mathbb{N}} X^n$. We then can define $(X^n)^m$, which is a tuple of tuples.

One can interpret this nonassociatively as *matrices* and write $X^{n \times m}$ instead, which is the set of $n \times m$-matrices over X.

A *relation* $\sim$ over a set X is a subset of $X \times X$. It is called an *equivalence relation* if it is reflexive, symmetric, and transitive. We write $x \sim y$ if $x, y \in X$ are in relation. In case of an equivalence relation, the *equivalence class* an element $x \in X$ belongs to is denoted by $[x]_\sim$. The set of equivalence classes induces a partition of X. The set of equivalence classes is addressed by $X/{\sim} = \{[x]_\sim \mid x \in X\}$. This set is also called a *quotient* of X.

For a linear order $(I; \leq)$ by $(X_i)_{i \in I}$ we denote a family that is the sequence of X_i for all $i \in I$ ordered by $\leq$. Usually, we have the case that $I = \mathbb{N}$.

A subset $f \subseteq X \times Y$ is called a (partial) *function* or a *map* if for all $x \in X$ there exists at most one $y \in Y$ such that $(x, y) \in f$. Then we also write $f(x) = y$ or $x \mapsto y$ and $f \colon X \to Y$ to indicate that the function maps from X to Y. A function $g \colon X' \to Y'$ is the *extension* of a function $f \colon X \to Y$ if $X \subseteq X'$, $Y \subseteq Y'$ and $f \subseteq g$. If for all $x \in X$ there exists exactly one $y \in Y$ such that $(x, y) \in f$, we call f a *total function*. Note that functions of higher arity come into being by choosing X to be a Cartesian product. The set of all functions of the form $X \to Y$ is denoted as Y^X. Given two functions $f \colon X \to Y$ and $g \colon Y \to Z$, by $g \circ f$ we denote the *composition* of f and g, which is defined as $\{(x, z) \in X \times Z \mid g(f(x)) = z\}$. For a function $f \colon X \to X$ and $i \in \mathbb{N}$ we may write f^i to express i compositions of f. For a function $f \colon X \to Y$ the *inverse function* $f^{-1} \colon Y \to 2^X$ is defined as $\{(y, \tilde{X}) \in Y \times 2^X \mid \tilde{X} = \{x \mid f(x) = y\}\}$. A function can be *injective* or *surjective*. If it is both, it is *bijective*. If f is injective, we may assume that the inverse function is a (partial) function $Y \to X$. If f is surjective, then f^{-1} is total.

Structure preserving mappings between algebraic objects, e.g. like groups, are called *homomorphisms*; exact definitions are provided later. Injective homomorphisms are called *monomorphisms*, surjective homomorphisms are called *epimorphisms*, and bijective homomorphisms are called *isomorphisms*. A homomorphism $X \to X$ is called an *monomorphism* and bijective monomorphisms are called *automorphisms*.

We use the Landau symbol $\mathcal{O}$, where $\mathcal{O}(f)$ contains all functions that do not grow faster as f, neglecting constant factors.

2.2 Structures

We are interested in computations whereas a computation usually needs an input and a computing device. In this chapter we look at inputs in a very general way. An input is a structured piece of information. For example, the input could be a word, which is a linear order on some pieces of information. Other important examples are graphs and trees.

All these different kinds of inputs can be summarized by the notion of structures. Structures are important subjects in the study of model theory. Structures are basically collections of relations. This is a more general notion compared to algebras where the relations are functions. We will cover algebras later. Although they are structures, we use them in a totally different way.

We loosely use the notation of the book [Alm94].

Definition 1 (Structure)**.** *A structure is a pair* $(\mathbb{D}; R)$ *where* $\mathbb{D}$ *is a set and* $R = (R_i)_{i \in [r]}$ *is a family of* r *relations over* $\mathbb{D}$*, so* $R_i \subseteq \mathbb{D}^{\alpha(i)}$*, where* $\alpha(i)$ *is the arity of* R_i*.*

It is also possible to allow for more than one domain set, i.e. to consider many-sorted structures. In the context of input structures we have no requirement for this feature, however, the algebras we define later incorporate it.

Each structure has a signature. A signature captures the format of a structure. Later when we introduce logic we need this notion in order to ensure compatibility of structures and logic formulas.

Definition 2 (Signature of a structure)**.** *A signature, usually denoted as* σ*, is an element of* $\mathbb{N}^*$*. Given a structure* $S = (\mathbb{D}; R)$ *with relations* $R = (R_i)_{i \in [r]}$*, the signature of* S *is defined as* $\sigma(S) = (\alpha(i))_{i \in [r]}$*.*

We see that a signature just captures the arities of the relations. Note that later we distinguish between signatures for different objects like structures and algebras because the arity of a function is one lower than the arity of it if interpreted as a relation.

Also notice that we did not require any properties for the domain. It could be infinite, which is an important case for many applications, however, we focus on finite input structures only.

In the following we consider typical examples for classes of structures. They all belong to distinct areas with unique notation and conventions. We try to unify all under the umbrella of structures while at the same time keeping the usual notation and conventions.

2.3 Graphs

Graphs come in different shapes. Basically, they consist of a set and a binary relation on this set.

Definition 3 (Graph)**.** *A structure with the signature* (2) *is called a (directed) graph. For a graph G we write $(V;E)$, where $E \subseteq V \times V$. If E is symmetric, then G is called undirected.*

A graph consists of *nodes* or, synonymously, *vertices*. They are connected by *edges*. Edges may have a direction. If they do, we say the graph is *directed*. In this case the edges are a binary relation between nodes as indicated by the definition. Undirected graphs can be modeled by enforcing the edge relation to be symmetric. If we do not want hooks in the graph, we also require antireflexivity. In the case of undirected graphs it is also common to define the edge set E as a set of sets of vertices of size two: $E \subseteq \binom{V}{2}$.

Given a graph G, by $V(G)$ we address the set of vertices of G and by $E(G)$ the set of edges. The empty graph we shortly denote by $\emptyset$.

A graph with a maximal set of edges is called *complete* or a *clique*. A complete graph of n vertices is denoted by K_n. A graph without edges is called *independent*. A graph G is called k-*partite* if $V(G)$ is the disjoint union of k partitions and there are no edges within a partition. If there are two sets making up the graph, we call it *bipartite*. A maximal bipartite graph where the two sets have size m and n, we address by $K_{m,n}$. A *star* is a bipartite graph of the form $K_{1,n}$.

Given two graphs G_1 and G_2, then $G_1 \cup G_2 := (V(G_1) \cup V(G_2); E(G_1) \cup E(G_2))$ and $G_1 \cap G_2 := (V(G_1) \cap V(G_2); E(G_1) \cap E(G_2))$. Further, given a graph G, the complement $\overline{G}$ contains exactly the edges that G does not have. For two graphs G_1 and G_2, $G_1 \subseteq G_2$ holds if $V(G_1) \subseteq V(G_2)$ and $E(G_1) \subseteq E(G_2)$.

For a graph $G = (V, E)$, a subset $V' \subseteq V$, and a subset $E' \subseteq E \cap V' \times V'$ we call (V', E') a *subgraph* of G. The graph $G[V']$ is called the *induced subgraph*, which has V' as vertex set and all edges of G that run inside V'. Given G, if we replace an edge $(u, v) \in E$ by two new edges (u, v') and (v', v) and add v' as a new node to the vertex set, we call the resulting graph a *subdivision* of G. Subdivisions of subdivisions of G are also subdivisions. If a graph G contains a subgraph S that is a subdivision of a graph M, then we call M a *minor* of G.

For a vertex v of some undirected graph, by $d(v)$ we denote the *degree* of v, which is the number of vertices u such that there is an edge from u to v. Here, we also say that v is a neighbor of u. In a directed graph a vertex v has an *in-degree* and an *out-degree* and the degree is the sum of in- and out-degree.

In a graph G, a *path* is a non-repeating sequence of neighboring vertices. A path where we allow for the first vertex to be equal to the last one is called a *cycle*. A graph in which for all vertices $u \neq v$ there exists a path from u to v is called *connected*. If in a graph G there exists a cycle containing all vertices of G, then the cycle as well as the graph are called *Hamiltonian*.

In many applications it is desirable to enrich graphs with more properties. If we regard graphs as structures that means introducing more relations next to the edge relation. For example, we may want to assign symbols to vertices. The relation capturing this is a subset of $V \times \Sigma$ where Σ is a set of symbols.

Often, we enforce that this new relation, in fact, is a map $V \to \Sigma$. However, it is convenient to represent this by using $|\Sigma|$ unary relations, i.e. the family $(Q_a)_{a\in\Sigma}$ and hence we get the structure $(V; E, (Q_a)_{a\in\Sigma})$. Even more restrictive than assigning unique symbols is the notion of *coloring*. Here, we refer to the symbols as *colors*. Usually, if we have k colors, we set $|\Sigma| = [k]$ and impose some condition on the coloring. A common one is requiring adjacent nodes to have different colors, but others exist as well depending on the application.

Another way to enrich graphs lies in embedding more precomputed information like the transitive closure of E. When we turn to logic this becomes important.

2.4 DAGs, Forests and Trees

An important type of graphs are directed acyclic graphs (DAGs). In these, one can define a notion of up and down, i.e. a partial order. DAGs will come up later again in the definition of circuits. Continuing further we arrive at the notions of *forests* and *trees*. We only consider finite forests and trees here.

There is a certain diversity when it comes to forests. The first version is the case where a forest is a special kind of undirected graph.

Definition 4 (Undirected forest and undirected tree)**.** *An undirected graph in which there exists at most one path between all pairs of vertices is called an undirected forest. An undirected graph in which there exists exactly exactly one path between all pairs of vertices is called an undirected tree.*

Equivalently, we can say that such a forest is a tree if it is connected. Usually one assigns a certain node in the graph the role of being the *root node*. In the undirected case, every node can act as a root node. In the course of this work it will always be either clear or unimportant which node should be considered the root, so we do not introduce an explicit notation. In contrast, in the directed case the root node is fixed.

Definition 5 (Directed forest and directed tree)**.** *Given an undirected tree $T = (V; E)$, let $r \in V$ be some node in T and let $d(r, v)$ be the length of the path from r to a node $v \in V$. Then $(V; E \cap \{(v_1, v_2) \mid d(r, v_1) < d(r, v_2)\})$ is a directed tree. Directed forests are unions of directed trees.*

In both the directed and undirected case, forests are unions of trees. In a directed forest, nodes of in-degree 0 are the root nodes. There is an isomorphism between directed trees and undirected trees that have a designated root node. The same then holds for forests, hence we will not strictly distinguish between both cases. Nodes of out-degree 0 are called *leaves*. The *depth* of a (directed) tree is the length of the longest path from the root to a leaf. The depth of a forest is the depth of its deepest tree. We call the members of a set of trees *balanced* if they satisfy that all leaves are of depth $\mathcal{O}(\log |V|)$. If a (directed) forest or tree has an edge $(u, v) \in E$, then u is the *parent* of V and v is a *child* of u. The children of a node are called *siblings*. By looking at the transitive closure of parents we get the set of *ancestors* which in turn defines an ancestor relation. It is the transitive closure of the edge set E.

The next concept is labeling. As for graphs in general, we can assign nodes a label. For a tree $(V; E)$, let Σ, which is also called alphabet, be a set of labels. Then a *labeling* is a total map $l\colon V \to \Sigma$. The labeled forest or tree can be denoted as $(V; E, l)$. For l we can also use the notation as introduced for general graphs and use a predicate Q_a for each letter.

Continuing, a forest can be ranked or unranked.

Definition 6 (Ranked labeled forest)**.** *Given a labeled forest $(V; E, l)$, it is called ranked if there exists a map $r\colon \Sigma \to \mathbb{N}$ such that for all $v \in V$, $r(l(v))$ is the number of children of v. The pair (Σ, r) is called the ranked alphabet.*

This definition implies an assignment of degree to label, since all nodes of the same label have the same degree. The notion of a ranked forest can also be applied to forests that are not labeled. We can regard these as ones in which all inner nodes have the same label. Hence all these nodes need to have the same degree.

Definition 7 (Ranked forest)**.** *A forest $(V; E)$ is called ranked if all non-leaf nodes have the same degree.*

For example, if in a ranked forest inner nodes have the degree two, it is called *binary*. Forests that are not ranked are called *unranked*, meaning that they are not necessarily ranked.

A ranked tree is called *complete* if all leaves are at the same depth.

Finally, there is a distinction of whether siblings have an order or not. In the way we defined forests and trees as graphs having certain properties, there is no order. The set of children of a node is simply a set. If we want to have a sibling order, we need to add it to the model. Where in a forest $F = (V; E)$ the set E resembles a kind of vertical order, a sibling order would be a horizontal one. So, given F, let $S \subseteq V \times V$ be an order relation that is total whenever it is restricted to a sibling set. Then an ordered forest can be denoted as $(V; E, S)$.

If we put some of the definitions together, we get trees that are directed, ranked, labeled, and ordered. Assume that $(V;E)$ is the directed graph, (Σ, r) the ranked alphabet, $l\colon V \to \Sigma$ the labeling function, and S the sibling order. We could denote such a tree as $(V;E,l,S)$, however most of the time we will not be that explicit. These kinds of trees will re-appear later in the form of terms. This is hinted by the fact that such trees can be easily denoted as terms. If a tree t has a root that is labeled a and has the subtrees $t_1, \ldots, t_{r(a)}$, then we can denote the tree as $\phi(t) = a(\phi(t_1), \ldots, \phi(t_{r(a)}))$, where ϕ gives us the translation. For leaves we get 0-ary function symbols. In the case of binary trees, the term notation can be made in-order instead of pre-order.

Similar to the previous case, the unranked variant is also relevant. Here, we can denote forests using an additional operation symbol that works as a horizontal concatenation of trees. This operation $+$ will be the topic of a later chapter. Here, we can already borrow it for notational purposes: If a tree t has a root labeled a, but is not ranked then we do not know the number of children. Since we are in the ordered case, the children can be regarded as a forest, or rather as a word of trees. Let f be this forest and $t_1, \ldots, t_n$ be the trees that make up f. Then we may write $f = t_1 + \ldots + t_n$ and $t = a(t_1 + \ldots + t_n)$.

DAGs occupy the space between trees and general graphs, but there are other notions, which also fit here. They all have the benefit of providing us with additional structure of the graph, which will help to solve problems more efficiently.

Graphs that are not trees can look quite like trees from a distance, which means that if you look closely, substructures may exist in a graph that are not a tree, but if the graph is abstracted away from those local substructures, what is left is a tree. It turned out that for computational purposes such graphs often behave as well as actual trees. This notion of abstraction goes under the name of *tree-decomposition*.

Definition 8 (Graph of tree-width k and tree-decomposition). *Given a graph $G = (V;E)$ then (T,τ) is called a tree-decomposition of G if T is a tree and $\tau\colon V(T) \to 2^V$ is a map for which the following conditions hold:*

- *For each $v \in V$ there exists $b \in V(T)$ such that $v \in \tau(b)$.*
- *For each $(u,v) \in E$ there exists $b \in V(T)$ such that $\{u,v\} \subseteq \tau(b)$.*
- *If there is a path from $r \in V(T)$ to $s \in V(T)$, then for all nodes $t \in V(T)$ on the path holds that $\tau^{-1}(r) \cap \tau^{-1}(s) \subseteq \tau^{-1}(t)$*

The elements of $V(T)$ are called bags. The size of the largest bag minus one is the width of the decomposition $\mathsf{width}(T,\tau)$. The minimal width of all decompositions of G is called the tree-width of G, which we denote as $\mathsf{width}(G)$.

Note that we define the width as the size of the largest bag *minus one*. That is because this way we get a tree-width of one for trees. A set $\mathcal{G}$ of trees is said to have bounded tree-width if there exists $k \in \mathbb{N}$ such that all graphs of $\mathcal{G}$ have a tree-width that does not exceed k.

Apart from tree-decomposition and width there exist two more general width concepts: Clique-width [CO00] and NLC-width [Wan94]. If a set of graphs has bounded tree-width, then it also has bounded clique-width. A set of graphs has bounded clique-width if and only if it has bounded NLC-width [CO00]. For those reasons the notions of bounded clique-width and bounded NLC-width are equivalent. In both cases a decomposition of a graph G is a term which, if evaluated, generates G. In the decompositions colorings are used. The minimal number of colors required then is the width.

Definition 9 (Graph of NLC-width k and NLC-decomposition)**.** *Graphs of NLC-width k are defined inductively:*

- *Colored graphs of a single node are graphs of NLC-width k.*
- *Given a graph $(V; E, l)$ of NLC-width k and a map $l' \colon [k] \to [k]$ then $(V; E, l' \circ l)$ has also NLC-width k.*
- *Given two disjoint graphs $G_1 = (V_1; E_1, l_1)$ and $G_2 = (V_2; E_2, l_2)$ of NLC-width k and a set $S \subseteq [k] \times [k]$, then $G_1 \times_S G_2$ has also width k, where $G_1 \times_S G_2$ is defined as $(V_1 \cup V_2; E_1 \cup E_2 \cup E', l')$,*

  $$E' = \{\{v_1, v_2\} \mid \exists (i,j) \in S \colon v_1 \in l_1^{-1}(i) \wedge v_2 \in l_2^{-1}(j)\},$$

  $l'(v) = l_1(v)$ if $v \in V_1$, and $l'(v) = l_2(v)$ if $v \in V_2$.

An NLC-decomposition of a graph G is a term made up of the previous operations that evaluates to G.

Definition 10 (Graph of clique-width k and clique-decomposition)**.** *Graphs of clique-width k are defined inductively:*

- *Colored graphs of a single node are graphs of clique-width k.*
- *Given two disjoint graphs $G_1 = (V_1; E_1, l_1)$ and $G_2 = (V_2; E_2, l_2)$ of clique-width k then $G_1 \cup G_2$ has also clique-width k where $G_1 \cup G_2 = (V(G_1) \cup V(G_2); E(G_1) \cup E(G_2), l_1 \cup l_2)$, i.e. $l(v) = l_1(v)$ if $v \in V(G_1)$ as well as $l(v) = l_2(v)$ if $v \in V(G_2)$.*
- *Given a graph $(V; E, l)$ of clique-width k and two colors $a, b \in [k]$, then $l_{a \to b}(V)$ is defined as $l_{a \to b}(v) = l(v)$ if $l(v) \neq a$ and $l_{a \to b}(v) = b$ else. Now, $(V; E, l_{a \to b})$ has also clique-width k.*

- *Given a graph $(V; E, l)$ of clique-width k and two colors $a, b \in [k]$, then the edge set $E_{a,b}$ is defined as $E \cup \{\{u, v\} \mid l(u) = a,\ l(v) = b,\ u \neq v\}$. Then $(V; E_{a,b}, l)$ is also a graph of clique-width k.*

A clique-decomposition of a graph G is a term made up of the previous operations that evaluates to G.

2.5 Words

Words are trees, trees are graphs, and graphs are structures. A word corresponds to a degenerated tree, which is a list. Hence, we can regard the base set of words as numbers, for which the order predicate $<$ exists. Together with a set of symbols assigned to the positions this yields that a word of length n is a structure $([n]; <, (Q_a)_{a\in\Sigma})$. Notice that $<$ is actually the transitive closure of a local comparison predicate $+1$. An equivalent way to conceptualize words is as maps of the form $[n] \to \Sigma$. Infinite words we get through $\mathbb{N} \to \Sigma$. Most commonly, a word w is seen just as a sequence written as $w = w_1 w_2 \ldots w_n$, which is short for the tuple $(w_1, w_2, \ldots, w_n)$. Usually, we require Σ to be finite and non-empty. If that is the case, we call Σ an *alphabet* and its elements *letters*.

Concatenation is a binary operation taking two values x and y where the result is the tuple (x, y). We usually write xy instead. Also, if $x = x_1 x_2$ and $y = y_1 y_2$ are also already words, we do not consider xy to be a word of words, but $xy = x_1 x_2 y_1 y_2$, i.e. concatenation is associative. That way each word can be decomposed until we reach non-decomposable elements, which are the letters.

Given an alphabet Σ, by Σ^* we denote the set of all finite words we can make up of letters in Σ. The *empty word* is denoted by ϵ and contained in Σ^*. Note that Σ^* together with concatenation is a free monoid, where ϵ is the neutral element.

The empty word has length 0. A letter considered as a word has length 1. Given a word $w = uv$, by $|w|$ we denote the length of w, which is defined as $|w| = |u| + |v|$. By Σ^n we denote all words of length n. Given a word $w \in \Sigma^*$ and $X \subseteq \Sigma$, by $|w|_X$ we denote the number of letters in w in X. For $a \in \Sigma$, we set $|w|_a = |w|_{\{a\}}$. Also, for $i \in [n]$, by $w(i)$ we address the i-th letter of the word. For convenience we also write w_i sometimes. Given a word $w \in \Sigma^*$, then $u \in \Sigma^*$ is called a *prefix* of w if there exists $v \in \Sigma^*$ such that $uv = w$. Similarly, if v exists so that $vu = w$ then u is a *suffix* of w. If u is a the prefix of a suffix of w, then w is called a *factor* or *infix* of w.

As for graphs and trees we again have the means to enrich the structure representing a word. One example that we will cover in the next section, is, where a word actually has a tree structure hidden within it. By exposing this tree structure many problems become tractable in contrast to the general case, which we may call the context-free case.

There are different kinds of additional predicates we can add to the pure word model. One kind consists of *numerical predicates*. In general, a numerical predicate can be purely defined through the domain. An example is the $<$ predicate over $[n]$ as well as the ternary relations $+$ or $\times$. Here, for example, $(i, j, k) \in +$ if $i + j = k$. The predicates $(Q_a)_{a\in\Sigma}$, which assign the letters, are not numerical. They introduce additional information. One can see the presence of numerical predicates as precomputed information obtained purely by the domain. In the next section we consider a setting of words enriched by a certain non-numerical predicate.

2.6 Nested and Well-Matched Words

There are many approaches to bring together trees and words. This is desired as some computational models take words as inputs, where at the same time the input is supposed to resemble a tree. We focus on one particular approach that was conceived of in the context of visibly pushdown languages. The model has two equivalent incarnations: Nested words and well-matched words. A *nested word* is a word with an additional binary relation $\rightsquigarrow$, so $w = ([n]; <, (Q_a)_{a\in\Sigma}, \rightsquigarrow)$ that satisfies the following:

- $i \rightsquigarrow j$ implies $i < j$.
- $i \rightsquigarrow j$, $i' \rightsquigarrow j'$, and $i < i'$ imply that $j' < j$ or $j < i'$.

Also, we require that each position is in relation with at most one other position. That means that this indeed is a nesting without crossings and we call $\rightsquigarrow$ a *matching predicate*.

A closely related model are *well-matched words*. Here, we partition the input alphabet Σ into three sets Σ_{call}, Σ_{ret}, and Σ_{int} and write $\hat{\Sigma} = (\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \Sigma_{\text{int}})$; $\hat{\Sigma}$ is called a *visible alphabet*. These sets contain call, return, and internal letters. Whenever we work with well-matched words, the partition of the alphabet is fixed. Now, w is a well-matched word if either it is empty, consists only of internal letters or it is of the form $w_1 a w_2 b w_3$, where $a \in \Sigma_{\text{call}}$, $b \in \Sigma_{\text{ret}}$, and w_1, w_2, w_3 are again well-matched words.

Thus, a well-matched word is just an ordinary word $([n]; <, (Q_a)_{a\in\Sigma})$ that satisfies an additional property with respect to $\hat{\Sigma}$.

The function $\Delta\colon \hat{\Sigma}^* \to \mathbb{Z}$ assigns words their height, which is the number of call letters minus the number of return letters: $w \mapsto |w|_{\Sigma_{\text{call}}} - |w|_{\Sigma_{\text{ret}}}$. For a word w, we call the sequence

$$\Delta(\epsilon), \Delta(w_1), \Delta(w_1 w_2), \ldots, \Delta(w_1 \ldots w_i), \ldots, \Delta(w)$$

the *height profile* of w.

There is an isomorphism between nested words and well-matched words. Given some nested word over Σ, we can build a corresponding well-matched word over $(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \Sigma_{\text{int}})$, where the three parts are copies of Σ. If in the nested word the letter in some position i is not part of a matching, we declare it to belong to Σ_{int} in the well-matched word. If it is part of a matching $i \rightsquigarrow j$, then we declare it to belong to Σ_{call}. Finally, if $j \rightsquigarrow i$ is the case, it belongs to Σ_{ret}. On the other hand, if we are given a well-matched word, we can compute the matching predicate. Doing so makes it obvious that $\rightsquigarrow$ can be interpreted as making the matching information directly available in contrast to well-matched words.

There exists also an isomorphism between unranked forests and nested words if we assume that all positions are part of a matching, which is equivalent to $\Sigma_{\text{int}} = \emptyset$ if seen from the perspective of well-matched words. This is a weak restriction since internal letters can be simulated by pairs of call and return letters. If we have a tree $t = a(f)$ that has a root labeled a and a forest $f = t_1 + \ldots + t_k$ as children, then we define the nested word $\mathsf{nw}(t) = a\mathsf{nw}(f)a$ where the first and the last positions match and $\mathsf{nw}(f) = \mathsf{nw}(t_1) \ldots \mathsf{nw}(t_k)$. Note that for a leaf we get $\mathsf{nw}(a()) = aa$. Conversely, given a nested word over alphabet Σ, we can define a forest over Σ^2 that resembles the nested word. Since there is an isomorphism between nested and well-matched words, we may write $\mathsf{wm}(t)$ to get the well-matched word for some tree, forest, or nested word t. To get the forest from nested or well-matched words, we write $\mathsf{forest}(\cdot)$. By $\mathrm{WM}(\hat{\Sigma})$ we denote the set of well-matched words with respect to $\hat{\Sigma}$. These two sets and the set of unranked forests are related in the way described.

Since both versions are isomorphic, we will use them interchangeably. However, notice that nested words possess more directly accessible information than well-matched words. In well-matched words some computation is needed to obtain the corresponding matching predicate. At the same time the way from nested words to well-matched words does not need actual computation. As we will see, this has consequences like the existence of a finite automaton model for nested words whereas well-matched words need a pushdown automaton model.

The model we described so far can be extended. Given Σ and a partition $(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \Sigma_{\text{int}})$ then only some words of Σ^* are well-matched words, so $\mathrm{WM}(\hat{\Sigma}) \subseteq \Sigma^*$. What meaning can we assign the other words? A word $baba \in \Sigma^*$ for $a \in \Sigma_{\text{call}}$ and $b \in \Sigma_{\text{ret}}$ has a matching in the second and third position, but the first and last positions are unmatched. We do not call *baba* well-matched, but we do call it *matched* in the context of a partition $(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \Sigma_{\text{int}})$. We can now just allow for unmatched positions and consider those positions pending matchings. The correspondence in terms of nested words can be made available by introducing a symbol ∞. The word *baba* becomes a structure of domain $\{1, 2, 3, 4\}$ and we have $2 \rightsquigarrow 3$, but further we get $-\infty \rightsquigarrow 1$ and $4 \rightsquigarrow \infty$. The symbols ∞ and $-\infty$ are not part of the domain, so we introduce unary relations $-\infty \rightsquigarrow$ and $\rightsquigarrow \infty$. Such structures we call *weakly nested words*. Besides different expressibility this model behaves differently in certain situations. For the most part we will stick to nested words and well-matched words as they offer nicer presentation, all expressibility we need, as well as the isomorphism to unranked forests. Note that in the literature both versions occur.

2.7 Conclusion

Summary

We introduced basic notation for the chapters ahead. Besides arranging the content we mainly gave a representation of well-established concepts. One reoccurring challenge lies in the fact that this work spans over many fields all having their unique and sometimes mutually exclusive notation.

In particular we looked at structures. Graphs, forests, trees, and words are special cases, which we also covered. Between graphs and trees there is some middle-ground in terms of decompositions.

Finally, we arrived at words. They are degenerated trees and at the base of computational complexity. Words are what we use as inputs and complexity is measured in the length of the input. To bridge words and trees we also looked into nested words and well-matched words.

Contributions

In this section we did not define any new objects or obtain new results, so the contribution lies in the presentation of what is already known.

Sources and Related Work

All objects we handle can be interpreted as structures; hence model theory is at the base. See e.g. the book of Ebbinghaus and Flum [EF95] for an introduction in finite model theory. Structures and algebras are closely related on the surface, but we use them in very different ways. An introduction to this area is, among others, the book of Almeida [Alm94]. Algebras will be covered in the next chapter.

Another special case of structures are graphs. A comprehensive treatment of graph theory can be found in the book of Diestel [Die12].

To link trees and graphs we looked at two kinds of decompositions: Tree-decompositions [Hal76] and clique-decompositions [CO00]. These decompositions can be used to assign a width to a graph. There also exists a decomposition variant that is equivalent to clique-width if bounded width is considered, which is NLC-width [Wan94]. A set of graphs has bounded clique-width if and only if it has bounded NLC-width [CO00]. There is a large body of research concerning those decomposition and width notions.

For word structures there are many sources for basics on words and word languages, for example the books of Hopcroft and Ullman[HU79] and Straubing [Str94]. Nested words and well-matched words were introduced by Alur and Madhusudan in [AM04, AKMV05, AM09]. They are rooted in visibly pushdown languages as introduced by Mehlhorn [Meh80] and popularized by Alur and Madhusudan [AM04].

Further Research

With regards to the way the different width concepts will be used later, it would be interesting and beneficial to lower the upper bound of deciding bounded clique-width or NLC-width. Up to now the upper bound is **P** and a desirable bound would be **NC**.

Chapter 3

Algebras

In the previous chapter we considered structures, which were motivated as a way to model data and inputs. Now, we need the means to do computations on inputs. To achieve this we introduce *universal algebras* (*algebras* for short) and then continue by covering terms, which can be evaluated over algebras, and homomorphisms, which are mappings between algebras.

3.1 Universal Algebras

An algebra itself happens to be a special kind of structure, however, algebras are used in a very different manner.

Definition 11 (Universal algebra)**.** *A universal algebra is a structure where each relation is a total function. The functions of an algebra are called operations. If not all functions are total, it is called a partial algebra.*

An algebra often has a domain, which can be regarded as containing structures of some kind. The operations of an algebra then can be used to combine and modify structures. An example is the free monoid whose domain contains words and its operation combines them.

In this setting, a signature of an algebra holds the arities of the operations of the algebra. For some algebra $\mathcal{A}$, its <u>*signature*</u> is $\sigma(\mathcal{A}) \in \mathbb{N}^*$. Note that there this is a difference compared to structures: The arity of a function is one less compared to its arity if interpreted as a relation; e.g. a constant function has arity 0, but if, in contrast, regarded as a relation it has arity 1.

There are situations where it is useful to have more than one domain set. This stems from the idea that elements belong to some type and operations may be sensitive to that type. Although multiple domain sets could be made into one by building the Cartesian product or a union, this approach is cumbersome, unnatural and usually leads to partial algebras. To that end we define many-sorted signatures and many-sorted algebras.

Definition 12 (Many-sorted signature)**.** *Let $S \in \mathbb{N}$ be a number of sorts. A many-sorted signature σ of k operations is an element of $([S]^* \times [S])^k$.*

A (many-sorted) signature is a k-tuple of pairs $\sigma = ((w_1, a_1), \dots, (w_k, a_k))$. The i-th pair (w_i, a_i) of the tuple codes the in- and outputs of the i-th operation. The first element w_i of the pair is a word. If this word has length $l = |w_i|$, the i-th operation has arity l. The letters of such a word indicates the sorts of the inputs of an operation. The letter a_i specifies the output type. We use the following notation:

- $\mathsf{In}_\sigma(i)$ addresses the word w_i. Also, $\mathsf{In}_\sigma(i,j)$ is the j-th letter of $\mathsf{In}_\sigma(i)$.
- $\mathsf{Ar}_\sigma(i)$ addresses the arity $|w_i|$.
- $\mathsf{Out}_\sigma(i)$ addresses the letter a_i.
- $\sigma(i)$ addresses (w_i, a_i).

In all notation defined we also allow for using an operation $\circledast$ instead of i, so e.g. $\sigma(\circledast)$ is the tuple (w, a) that corresponds to the i-th operation $\circledast$ of some algebra.

Single sorted signatures embed into this more general setting. In this case the words are unary and the only information left is the length of a word, which gives us the arity of an operation.

Definition 13 (Many-sorted universal algebra)**.** *Given a many-sorted signature σ with S sorts, a many-sorted universal algebra of signature σ is a tuple $\mathcal{A} = (\mathbb{D}; O)$ where:*

- *$\mathbb{D} = (\mathbb{D}_i)_{i \in [S]}$ is the domain set of S sorts. We call the sets $\mathbb{D}_i$ <u>subdomains</u>.*
- *$O = (\circledast_i)_{i \in [|\sigma|]}$ are the operations with*

$$\circledast_i \colon \mathbb{D}_{\mathsf{In}_\sigma(i,1)} \times \ldots \times \mathbb{D}_{\mathsf{In}_\sigma(i,\mathsf{Ar}_\sigma(i))} \to \mathbb{D}_{\mathsf{Out}_\sigma(i)}.$$

Sometimes by abuse of notation we will use $\mathbb{D}$ as if it was the union of all subdomains and also for convenience if $\mathcal{A}$ is an algebra, we will use $\mathcal{A}$ as if it was the domain set itself.

An algebra $\mathcal{A}$ is said to be *finitely generated* if there exists a finite subset G of the domain $\mathbb{D}$ such that all other values of $\mathbb{D}$ can be obtained using the operations of the algebra and constants in G, written as $\langle G\rangle = \mathcal{A}$. All algebras covered here are finitely generated. Also, we assume that all algebras $\mathcal{A}$ have a set of constant operations that generate $\mathcal{A}$. For example, we define the algebra of natural numbers with addition and multiplication as $(\mathbb{N}; +, \times, 0, 1)$ since $\langle 0, 1\rangle = \mathbb{N}$.

By writing $\langle G\rangle$ we generate all terms using the constants in G, which then evaluate to some value. Whether one should include the empty term or not is an issue. Hence, to make things easy, we disallow empty terms. If we still want an element in the domain that acts as the empty term, we include an additional constant operation explicitly. Usually, what is desired is not the empty term itself, but a neutral element, like in the case of monoids. Also, this makes sense because e.g. it is not directly clear what sort the empty term has.

Note that we do not allow finitary operations. Finitary operations would be practical in some cases, however, in other situations they lead to more complexity. It is possible to simulate finitary operations by using families of algebras. By doing so, we also have more control over the actual arity used and this concept fits well with families of circuits we will introduce later.

Definition 14 (Family of algebras)**.** *A family of algebras is defined as* $(\mathcal{A}_n)_{n\in\mathbb{N}}$*, where*

$$\mathcal{A}_i = ((\mathbb{D}_1)^{p_1(i)}, \ldots, (\mathbb{D}_S)^{p_S(i)}; \circledast^i_1, \ldots, \circledast^i_k)$$

and for $j \in [S]$*,* p_j *is some polynomial. Similarly, we define* $\circledast^i_j$ *through signatures and require:*

- $\mathsf{Out}_{\sigma(\mathcal{A}_1)}(j) = \mathsf{Out}_{\sigma(\mathcal{A}_i)}(j)$ *for all* $j \in [k]$ *and* $i \in \mathbb{N}$.
- *There exists a polynomial* p *such that for all* $i \in \mathbb{N}$ *and all* $j \in [k]$ *it holds that* $\mathsf{Ar}_{\sigma(\mathcal{A}_i)}(j) \leq p(i)$.

Given a family of algebras $\mathcal{A}$, we can naturally define a family of signatures $\sigma(\mathcal{A}) = (\sigma(\mathcal{A}_n))_{n\in\mathbb{N}}$.

Example 15. *We look at some algebras:*

- *Consider the rational numbers with multiplication, division, addition and subtraction. This does not form an algebra directly because division is a partial function. Hence, either we need one domain* $\mathbb{Q}$ *and one domain* $\mathbb{Q}\setminus\{0\}$ *together with some additional operations, or we use one domain* $\mathbb{Q}\cup\{\perp\}$ *where* $\perp$ *stands for an undefined value due to a division by* 0.
- *The positive rationals without subtraction form a valid algebra* $(\mathbb{Q}^+; +, \times, \div, 1)$.

- *The natural numbers together with the operations addition and multiplication form an algebra* $(\mathbb{N}; +, \times, 0, 1)$*. Finitary versions of* $+$ *and* $\times$ *can be simulated with a family of algebras* $(\mathbb{N}; +_n, \times_n, 0, 1)_{n\in\mathbb{N}}$ *where* $+_n$ *and* $\times_n$ *take* n *naturals as input.*

- *Natural numbers can be represented as binary words, i.e. there is an isomorphism between* $\mathbb{N}$ *and* $\mathbb{B}^*$*. The algebra* $(\mathbb{B}^*; +^{\mathbb{B}}, \times^{\mathbb{B}}, 0, 1)$ *has two binary operations, which add or multiply two binary words representing integers. The result is the sum or product without leading* 0*-s. Later it will be helpful if we fix the length of the binary inputs. We then get a family* $(\mathbb{B}^n; +^{\mathbb{B}}_n, \times^{\mathbb{B}}_n, 0^n, 0^{n-1}1)_{n\in\mathbb{N}}$ *where within one member of the family the length is fixed to* n*. To account for large values, all results are taken modulo* 2^n*.*

3.2 Terms

Terms (or, equivalently, *expressions*) are labeled and ranked trees that we can build from a signature. It can be regarded as a syntactic object to which we can assign meaning through an algebra of the same signature. For instance for the signature $\sigma = (2, 2, 0, 0)$ we can build a term $t = (((*_4 *_1 *_4) *_2 *_3) *_1 *_4)$, where $*_i$ is an abstract placeholders for operations. If we encounter a term, which we see in context of a concrete algebra, we write $\circledast_i$, which is the i-th operation of the algebra. Now, consider the algebra $(\mathbb{N}; +, \times, 0, 1)$ that has signature σ. We can evaluate t over $(\mathbb{N}; +, \times, 0, 1)$, which yields the value 1. One can now also write the term directly by using the operations, which leads to the representation $(((1 + 1) \times 0) + 1)$ for t. Evaluated over $(\mathbb{B}; \vee, \wedge, \bot, \top)$, we get $\top$ and the term can be written as $(((\top \vee \top) \wedge \bot) \vee \top)$. Note that terms are trees and the way we write them is just one of many to represent the tree. Therefore, it does not matter whether we use infix or postfix notation for that matter.

We not only consider terms that are purely composed out of operations, in which the leaves are 0-ary operations, but also terms with variables. We allow for a number of variables, which each may occur several times within a term. For example, if we look at terms with variables over the algebra $(\mathbb{B}; \wedge, \vee, \neg, \bot, \top)$, we can ask for valuations of the variables such that the term evaluates to $\top$. This problem is known as the **NP**-complete SAT problem [Coo71].

Definition 16 (Term)**.** *Given a signature* σ *of* S *sorts and a number of variables* n *together with a map* $\xi\colon [n] \to [S]$ *called a signature of the variables, then a* (σ, ξ)*-term is a labeled ranked ordered tree where* $(V; E)$ *is the graph of the tree, the labeling is* $l\colon V \to \{*_1, \dots, *_{|\sigma|}\} \cup \{\Box_1, \dots, \Box_n\}$*, and the following hold:*

- *If a node is labeled $*_i$, its rank is $\mathsf{Ar}_\sigma(*_i)$ and if it is labeled $\Box_i$, its rank is 0, i.e. it is a leaf.*
- *We say $v \in V$ has sort $\mathsf{Out}_\sigma(l(v))$ if $l(v) = *_i$ and otherwise it has sort $\xi(i)$ if $l(v) = \Box_i$.*
- *We require that if v_j is the j-th child of v, this implies that $\mathsf{Out}_\sigma(l(v_j)) = \mathsf{In}_\sigma(l(v), j)$.*

The set of (σ, ξ)-terms of variable signature ξ is denoted by $\mathbb{T}^\xi(\sigma)$ and for $s \in [S]$ the subset of all terms whose root has sort s is denoted by $\mathbb{T}^\xi_s(\sigma)$. If for the number of variables $n = 0$ holds, we speak of σ-terms and for the sets of such terms we write $\mathbb{T}_s(\sigma)$ and $\mathbb{T}(\sigma)$. The set of terms having arbitrary variables of consistent types is denoted by $\mathbb{T}^*_s(\sigma)$.

In the single-sorted case, $\mathbb{T}_s(\sigma)$ collapses to $\mathbb{T}(\sigma)$ and $\xi\colon [n] \to [1]$ can be regarded as just the number n, so we may write $\mathbb{T}^n(\sigma)$. We also allow to write $\mathbb{T}^X(\sigma)$ for some set X by assuming a bijection between $[n]$ and X.

A term in which for $i \in [n]$ the variable $\Box_i$ appears at most once is called *linear*. Note that $\mathbb{T}(\sigma) \subseteq \mathbb{T}^\xi(\sigma)$ and also note that variables can be interpreted as constant operations if considered over a more complicated algebra as well. We will go deeper into this later.

A (σ, ξ)-term we denote like trees as $t(x_1, \ldots, x_n)$. It is no coincidence that we chose as the way of denoting terms the same as the one we introduced for trees; terms are trees after all. In this notation, t is a tree which has n leaves $x_1, \ldots, x_n$.

A special case of terms with variables are contexts where we have one variable for every sort, i.e. $n = S$. However, a context must only contain one variable at most.

Definition 17 (Context)**.** *A (σ, ξ)-term is called a σ-context if $\xi\colon S \to S$ with $x \mapsto x$ and if there exists only one node labeled $\Box_i$ for some $i \in [n]$. This node is also called hole. The set of σ-contexts is denoted as $\mathbb{C}(\sigma)$. The set of contexts that only have a hole of sort s is denoted as $\mathbb{C}^s(\sigma)$. Contexts that evaluate to some sort s' are denoted as $\mathbb{C}_{s'}(\sigma)$.*

We will later come back to the notion of contexts.

For an operation $\circledast$ we defined $\sigma(\circledast)$, which holds the information of the input and output types. We also have ξ, which is the signature of the variables. In an analogous manner we can define $\sigma(t(x_1, \ldots, x_n)) \in [S]^n \times [S]$ for some $t(x_1, \ldots, x_n) \in \mathbb{T}^\xi(\sigma)$.

A *substitution* for a sort $i \in [n]$ is a function $\mathbb{T}^\xi(\sigma) \times \mathbb{T}^\xi_i(\sigma) \to \mathbb{T}^\xi(\sigma)$, which takes terms $t(x_1, \ldots, x_n)$ and $t'(x_1, \ldots, x_n)$ and replaces each occurrence of $\Box_i$ in $t(x_1, \ldots, x_n)$ by $t'(x_1, \ldots, x_n)$. The result we denote as

$t(x_1, \dots, x_n)[x_i/t'(x_1, \dots, x_n)]$, which is again a term $t''(x_1, \dots, x_n)$ of the same sort as t.

Here, we introduced terms as trees; they can be thought of as inductively defined objects: Constants are terms and everything that is a term can be combined with other terms resulting in a term. So, actually, an operation $*_i$ takes terms and results in a term. This again forms an algebra:

Definition 18 (Term algebra)**.** *Given a signature σ of S sorts, the term algebra $\mathcal{T}(\sigma)$ is defined as $\big(\mathbb{T}(\sigma); (*_i)_{i\in[S]}\big)$ where $*_i(t_1, \dots, t_{\mathsf{Ar}_\sigma(i)})$ is the tree whose root is labeled $*_i$ and the children are $t_1, \dots, t_{\mathsf{Ar}_\sigma(i)}$, providing that the result is a valid term.*

We defined terms relative to some signature. A term can then be evaluated over algebras of the same signature. Here, each symbol $*_i$ in the term is interpreted as the i-th operation $\circledast_i$ of the algebra we evaluate in.

To also cover terms with variables, we need variable valuations to be able to evaluate such terms. A <u>*variable valuation*</u> is a map $\nu\colon [n] \to \mathbb{D}$.

Definition 19 (Evaluation of terms)**.** *Given is an algebra $\mathcal{A} = \big((\mathbb{D}_i)_{i\in[S]}, (\circledast_i)_{i\in[|\sigma|]}\big)$ of signature σ and S sorts, a term $t \in \mathbb{T}^\xi(\sigma)$ where $\xi\colon [n] \to [S]$ is the variable signature, and variable valuation map $\nu\colon [n] \to \mathbb{D}$ respecting ξ. The evaluation function $\mathrm{eval}^\nu_\mathcal{A}\colon \mathbb{T}^\xi(\sigma) \to \mathbb{D}$ is defined inductively:*

- *If t is of the form $*_i$, then $\mathsf{Ar}_\sigma(i) = 0$ and we set $\mathrm{eval}^\nu_\mathcal{A}(t) = \circledast_i$.*
- *If t is of the form $\square_i$, then we set $\mathrm{eval}^\nu_\mathcal{A}(t) = \nu(i)$.*
- *If t is of the form $*_i(t_1, \dots, t_j)$ for terms $t_1, \dots, t_j$, then $\mathrm{eval}^\nu_\mathcal{A}(t) = \circledast_i(\mathrm{eval}^\nu_\mathcal{A}(t_1), \dots, \mathrm{eval}^\nu_\mathcal{A}(t_j))$.*

For terms without variables, $\mathrm{eval}_\mathcal{A}$ is defined as $\mathrm{eval}^\emptyset_\mathcal{A}$.

Note that a term t over σ evaluated over $\mathcal{T}(\sigma)$ is again t. Also note that in order to get a non-empty term algebra $\mathcal{T}(\sigma(\mathcal{A}))$, it is necessary that $\mathcal{A}$ contains 0-ary operations, as we argued before by requiring a generating set to be present as constants. We see that every element that is generated by the generators has a term using the generators as constants.

Of course, we can also define evaluation for (σ, ξ)-terms without an valuation of the variables. A term with n variables then evaluates not to an element of $\mathbb{D}$ but to a function $\mathbb{D}^n \to \mathbb{D}$.

3.3 Homomorphisms

A homomorphism is a mapping between algebras that preserves structure, i.e. homomorphisms are distributive over the algebra operations.

We define homomorphisms in a very general way as maps between arbitrary many-sorted algebras. The notion to be found in [Alm94] for example, utilizes the restriction that homomorphisms only map between single-sorted algebras of the same signature. In that setting the i-th operation of an algebra is mapped onto the i-th of another, but since we map between arbitrary algebras we relax this restriction and allow an operation to be mapped onto a term with variables that has the same signature as the operation.

Definition 20 (Generalized homomorphism)**.** *Given two many-sorted algebras $\mathcal{A} = (\mathbb{D}; O)$ and $\mathcal{B} = (\mathbb{E}; P)$ of S, respectively S', sorts, as well as a map $\alpha\colon O \to \mathbb{T}^*(\sigma(\mathcal{B}))$ such that there exists a unique map $\beta\colon [S] \to [S']$ with $\sigma(\alpha(\circledast)) = \beta(\sigma(\circledast))$ for all $\circledast \in O$. Then $\alpha^*\colon \mathbb{T}(\sigma(\mathcal{A})) \to \mathbb{T}(\sigma(\mathcal{B}))$ is defined as follows: If $d, d_1, \ldots, d_{\mathsf{Ar}_\sigma(\circledast)} \in \mathbb{D}$ with $d = \circledast(d_1, \ldots, d_{\mathsf{Ar}_\sigma(\circledast)})$, then*

$$\alpha^*(d) = \alpha(\circledast)(\alpha^*(d_1), \ldots, \alpha^*(d_{\mathsf{Ar}_\sigma(\circledast)})).$$

We set $\phi\colon \mathbb{D} \to \mathbb{E}$ for $d \in \mathbb{D}$ to be $\phi(\mathrm{eval}_{\mathcal{A}}(t)) = \mathrm{eval}_{\mathcal{B}}(\alpha^*(t))$ if α^* satisfies that $\phi(\mathrm{eval}_{\mathcal{A}}(t)) = \phi(\mathrm{eval}_{\mathcal{A}}(t'))$ for all $t, t' \in \mathbb{T}(\sigma(\mathcal{A}))$ with $\mathrm{eval}_{\mathcal{A}}(t) = \mathrm{eval}_{\mathcal{A}}(t')$. The mapping ϕ then is called a generalized homomorphism.

The previous definition allows images of α to be non-linear terms. Usually, we only consider generalized homomorphisms where the images of α are linear terms unless stated otherwise. For mappings between algebras we also like to write $\mathcal{A} \to \mathcal{B}$, where, in fact, it is a mapping between domains, of course.

The classical notion of homomorphisms is a special case of the generalized homomorphisms we defined. In it, one assumes that source and target of the mapping have the same signature.

Definition 21 (Homomorphism)**.** *For two algebras $\mathcal{A}$ and $\mathcal{B}$ of the same signature, a generalized homomorphism $\phi\colon \mathcal{A} \to \mathcal{B}$ that is defined through $\alpha\colon O \to \mathbb{T}(\sigma)$ is a homomorphism if α restricted on non-constant operations is the identity map, i.e. α maps the i-th operation of $\mathcal{A}$ to the i-th operation of $\mathcal{B}$.*

As an example, consider the case of free monoids. Let $\phi\colon A^* \to A^*$ be a homomorphism. A free monoid has a binary operation that is concatenation and a homomorphism does not change that: $\phi(u \cdot v) = \phi(u)\phi(\cdot)\phi(v) = \phi(u) \cdot \phi(v)$. Further, we may assign arbitrary terms to constant operations, or in this case more precisely, words. So, we could have $a \mapsto abc$ for $a, b, c \in A$. In fact, it is known that e.g. in the special case of semigroups a homomorphism is determined by its images of the generating elements of the semigroup. In our case of generalized homomorphisms it is determined by α, that is the image of all operations. The group case embeds through the fact that the generating elements should be present as constant operations.

From now on we will also not strictly distinguish between generalized homomorphisms and homomorphisms as it is usually clear from the context.

We say an algebra $\mathcal{A}$ *divides* $\mathcal{B}$ if there exists a subalgebra $\mathcal{B}'$ of $\mathcal{B}$ and an epimorphism $\phi\colon \mathcal{B}' \twoheadrightarrow \mathcal{A}$; we write $\mathcal{A} \prec \mathcal{B}$. Given homomorphisms ϕ and ψ then ψ is said to *factor through* ϕ if there exists a homomorphism θ such that $\theta \circ \phi = \psi$.

Given a single-sorted algebra $\mathcal{A} = (\mathbb{D}; O)$, a congruence relation $\sim$ on $\mathcal{A}$ is an equivalence relation on $\mathbb{D}$ satisfying that for all $\circledast \in O$ of arity k and $x_i \sim y_i$ for $x_i, y_i \in \mathbb{D}$ and $i \in [k]$, holds that $\circledast(x_1, \dots, x_k) \sim \circledast(y_1, \dots, y_k)$. Such a relation induces as natural homomorphism $\pi_\sim\colon \mathbb{D} \to \mathbb{D}/{\sim}$ with $x \mapsto [x]_\sim$. Also, we can define $O/{\sim}$ from O. For $\circledast \in O$ there is $\circledast_\sim \in O/{\sim}$ such that $\circledast_\sim(\pi_\sim(x_1), \dots, i(x_n)) = \pi_\sim(y)$ if $\circledast(x_1, \dots, x_n) = y$. The induced algebra $\mathcal{A}/{\sim} = (\mathbb{D}/{\sim}; O/{\sim})$ is called the *quotient algebra*. The concept of congruences carries over to the many-sorted case.

Given a many-sorted algebra $\mathcal{A} = (\mathbb{D}; O)$, a *congruence relation* $\sim \subseteq \mathbb{D}_s \times \mathbb{D}_s$ on $\mathcal{A}$ is an equivalence relation on one of the subdomains of $\mathbb{D}$ that has to satisfy that if $x \sim y$ implies that for all terms $t \in \mathbb{T}_s(\sigma(\mathcal{A}))$ that have subterm t' with $\mathrm{eval}^s_{\mathcal{A}}(t') = x$ it holds that $\mathrm{eval}^s_{\mathcal{A}}(t) \sim \mathrm{eval}^s_{\mathcal{A}}(t[t'/t''])$ where t'' is some term with $\mathrm{eval}^s_{\mathcal{A}}(t'') = y$. Given a congruence on $\mathbb{D}_s$, we can define a quotient algebra $\mathcal{A}/{\sim}$. Here, the s-th subdomain becomes $\mathbb{D}_s/{\sim_s}$. However, we also have to build quotients in other subdomains as well. Given $\sim_s$ we define $\sim_i$ for all $i \in [S]$. For $x, y \in \mathbb{D}_i$ we let $x \sim_i y$ if for all terms $t \in T(\sigma(\mathcal{A}))_s$ that have subterm t' with $\mathrm{eval}^i_{\mathcal{A}}(t') = x$ it holds that $\mathrm{eval}^s_{\mathcal{A}}(t) \sim \mathrm{eval}^s_{\mathcal{A}}(t[t'/t''])$ where t'' is some term with $\mathrm{eval}^i_{\mathcal{A}}(t'') = y$. The quotient algebra $\mathcal{A}/{\sim}$ now is $((\mathbb{D}_1/{\sim_1}, \dots, \mathbb{D}_S/{\sim_S}); O')$, where O' contains quotients of the functions in O as defined for the single-sorted case. The natural homomorphism $\pi_\sim$ is also defined similarly as in the single-sorted case.

We defined a homomorphism based on a congruence. The converse is also possible: Given a homomorphism $\phi\colon \mathcal{A} \to \mathcal{B}$, the kernel is

$$\ker(\phi) = \{(a, b) \in \mathbb{D} \times \mathbb{D} \mid \phi(a) = \phi(b)\},$$

where $\mathbb{D}$ is the domain of $\mathcal{A}$.

Proposition 22. *Kernels of homomorphisms are congruence relations.*

Proof. Given a homomorphism $\phi\colon \mathcal{A} \to \mathcal{B}$ between two many-sorted algebras, the kernel $\ker(\phi)$ has to satisfy that $(\mathrm{eval}^s_{\mathcal{A}}(t), \mathrm{eval}^s_{\mathcal{A}}(t[t'/t''])) \in \ker(\phi)$ where $t, t', t'' \in \mathbb{T}(\sigma(\mathcal{A}))$, t' is a subterm of t and $(\mathrm{eval}^s_{\mathcal{A}}(t'), \mathrm{eval}^s_{\mathcal{A}}(t'')) \in \ker(\phi)$. Observe now that

$\phi(t) = \phi(t[t'/t''])$, i.e. t and $t[t'/t'']$ are in relation, so the condition for a congruence relation is satisfied. □

Theorem 23 (Homomorphism Theorem for many-sorted algebras)**.** *Given a homomorphism $\phi\colon \mathcal{A} \to \mathcal{B}$. Then there exists exactly one homomorphism $\psi\colon \mathcal{A}/\ker(\phi) \to \mathcal{B}$ such that $\phi = \psi \circ \pi_{\ker(\phi)}$ where $\pi_{\ker(\phi)}\colon \mathcal{A} \to \mathcal{A}/\ker(\phi)$. Also, if ϕ is an epimorphism, then ψ is an isomorphism.*

Proof. Let p_a be some element in the non-empty set $\pi^{-1}_{\ker(\phi)}(a)$ for $a \in \mathcal{A}/\ker(\phi)$. We choose $\psi\colon \mathcal{A}/\ker(\phi) \to \mathcal{B}$ as $a \mapsto \phi(p_a)$. Since $\phi(x) = \psi(\pi_{\ker(\phi)}(x))$ must hold, clearly ψ is unique. Also, observe that ψ is a homomorphism. If ψ is an epimorphism, $\psi^{-1}(x) \neq \emptyset$ for $x \in \mathcal{B}$. For $a \neq b$ we have that $\psi(a) \neq \psi(b)$ since otherwise $(p_a, p_b) \in \ker(\phi)$, which is a contradiction that shows isomorphism. □

An important notion is freeness. The name stems from the intuition that an algebra is free of nontrivial equations. In a commutative group, for example, the equation $xy = yx$ holds. We do not define equations formally here since this is out of scope. A standard way to define freeness is to say that a free algebra contains all terms, so we just consider the term algebras $\mathcal{T}(\sigma)$ as the free algebra with respect to a signature σ.

Freeness also exists for special kinds of algebras like monoids, groups, or even commutative groups. Those examples are clearly not free algebras, but they are free with respect to some *variety*.

Finally, we want to argue that the framework of many-sorted algebras and generalized homomorphisms that we laid out behaves as well as the classical notions. In the course of this work it will become apparent that these generalized objects indeed are useful and even necessary for certain applications.

3.4 Semigroups and Semirings

Previously, we considered algebras in a very general sense. Classical algebra, however, focuses on certain classes of algebras like groups or rings. This setting can, of course, be embedded in the framework we laid out before, however, this leads to some uncommon notation due to the generality we provided. When we deal with well-known objects like groups etc., we will use the common notation. For example, a group is noted as $(G; \cdot)$ whereas in our framework, the generators should be present as constant operations and the inversion as a unary operation.

First, we look at algebras having $\cdot$ as their single binary operation. Depending on properties of this operation we give these algebras different names. Such an algebra is called a magma if the operation is total, i.e. for all $x, y \in \mathbb{D}$ the product $x \cdot y$ is defined. An element $e \in \mathbb{D}$ is called *neutral* if for all $x \in \mathbb{D}$ holds that $x \cdot e = e \cdot x = x$. A magma with neutral element is called *unital magma*. A magma that is associative is called *semigroup*, i.e. for all $x, y, z \in \mathbb{D}$ holds that $(x \cdot y) \cdot z = x \cdot (y \cdot z)$. Terms over semigroups are trees, but because of associativity, we may drop the parentheses and hence get just a sequence. We also drop the $\cdot$ and just write xy for the product. A semigroup with a neutral element is called a *monoid*. A monoid in which each element has an inverse is called a *group*, i.e. for all $x \in \mathbb{D}$ there exists $x^{-1} \in \mathbb{D}$ such that $xx^{-1} = x^{-1}x = e$. A group that is commutative is called an *Abelian group*, i.e. for all $x, y \in \mathbb{D}$ holds that $xy = yx$. An element $x \in \mathbb{D}$ is called *idempotent* if $xx = x$.

Semigroups and monoids only differ in the presence of a neutral element. In application settings like ours, most authors decide to either go with monoids or with semigroups. For example, if a semigroup is used to recognize a language, then the empty word has to be left out if the semigroup is not a monoid. In this work we will mostly use monoids. A monoid M is called *free* over A if there exists a map $\iota\colon A \hookrightarrow M$ such that for all monoids N and maps $f\colon A \to N$ there exists precisely one homomorphism ϕ such that $\phi|_A \circ \iota = f$. It is then equivalent to A^*, which contains all sequences of elements in A including the empty one, so every free monoid is isomorphic to a monoid A^*. The elements of A^* can be regarded word structures. Further, monoid homomorphisms are already defined by the image of A.

We call an element x *absorbing* or a *zero* if for all $y \in \mathbb{D}$ holds that $yx = xy = x$. A zero is always unique. We call an element $x \in \mathbb{D}$ *nilpotent* if there exists a number $n \in \mathbb{N}$ such that $x^n = 0$ and x is called *aperiodic* if there exists a $n \in \mathbb{N}$ such that $x^n = x^{n+1}$. A semigroup is called *aperiodic* if all its elements are aperiodic. This is equivalent to saying that there is no subset of the semigroup that is a non-trivial group. A monoid homomorphism $\phi\colon A^* \to S$ is called *quasiaperiodic* if for all $t \in \mathbb{N}$ the image $\phi(A^t)$ does not have subsets that are non-trivial groups.

A *semiring* $(R; +, \cdot)$ is an algebra having two binary operations such that $(R; +)$ is a commutative monoid, $(R; \cdot)$ is a monoid, and distributivity holds, i.e. $(a+b)c = ac+bc$ as well as $c(a+b) = ca+cb$. If in addition $(R; +)$ is a group, $(R; +, \cdot)$ is a *ring*.

3.5 Conclusion

Summary

In this chapter we introduced the basic notions of algebra. The term algebra is very generic and used in many different ways throughout the literature. Here, we built upon the framework of universal algebras where a domain and functions over this domains are given. First, we defined many-sorted algebras as a way to have a clean distinction between different kinds of data and thereby bypass partial algebras. We also considered families of algebras.

After introducing basic algebra definitions, we continued with the concept of terms, which are trees that capture calculations within an algebra.

We then examined (generalized) homomorphisms. Again, we defined objects that are more general as usual. Still, one can consider them as being quite natural, and we will see that they are indeed needed later. We do not know of any publications containing this concept. We showed some results that are known from group theory and the theory of universal algebras, most importantly a generalized version of the Homomorphism Theorem.

Finally, we considered classical algebra objects like groups and rings in the context of our framework.

Contributions

We provided very general definitions of objects that usually occur in more specialized versions. We exhibited a framework consisting of families of many-sorted algebras, fitting congruence relations, generalized homomorphisms and a Homomorphism Theorem.

In [KLL17a, KLL17b] we already gave early versions of the definitions but enhanced some of them.

Sources and Related Work

We used the book of Almeida [Alm94] as a foundation and generalized notions from there. Related notions of many-sorted algebras can be found in [Cou90, Wir90, EM85], however while employing infinitely many sorts and infinite signatures.

There are different related theories. One of them is category theory itself, as well as an approach in which categories themselves act as a substitute for algebras [Til87]. Another related area could be type theory. For both category and type theory we did not dwell into research of finding similarities but suspect that both could be alternative frameworks to formulate results in. For this work the usage of algebras emerged naturally. Although not explicitly named as such, forest algebras [BW08], which we will cover in the next chapter, are an example of many-sorted algebras that we use extensively in this work.

Further Research

We performed first steps in showing the utility of this very general set of definitions. They could have more applications and they themselves can be researched more

Chapter 4

Recognition by Algebras

In this and the following chapters we consider recognition of languages. We have two ways to look at what languages are. On the one hand, a language is a subset of some (free) algebra. On the other hand it is a set of structures that share the same signature and maybe also other properties. A finite word, for example, can be interpreted as an element of a free monoid or as a structure whose domain consists of the word positions. In this chapter we examine recognition by algebra, so we regard languages as subsets of algebras. However, there is a one-to-one correspondence between the two perspectives.

Language recognition by algebras is best known in the form of monoids recognizing word languages. Here, a language L is a subset of Σ^*, i.e. a subset of the free monoid, and a monoid M recognizes a language L if there exists a homomorphism $\phi\colon \Sigma^* \to M$ such that $L = \phi^{-1}(\phi(L))$. In this case, intuitively speaking, the monoid M captures all the relevant information of L, or one could say that L, which in general is an infinite subset of an infinite monoid Σ^*, can be represented by $\phi(L)$ as a subset of M. If M is finite, we have a true compressed representation. In the case that M is finite, L is, in fact, regular.

We now want to take this mechanism and generalize it to arbitrary algebras.

Definition 24 (Recognition by algebras)**.** *Given the possibly many-sorted algebras $\mathcal{A} = (\mathbb{D}; O)$ and $\mathcal{B} = (\mathbb{E}; P)$, a language $L \subseteq \mathbb{D}$ of a single sort and a homomorphism $\phi\colon \mathcal{A} \to \mathcal{B}$, then L is recognized by $\mathcal{B}$ and ϕ if $\phi^{-1}(\phi(L)) = L$.*

An equivalent notion lies in saying that there is a subset X of $\mathcal{B}$ such that $\phi^{-1}(X) = L$. That means we find a corresponding set of L in $\mathcal{B}$, so L is embedded in $\mathcal{B}$. Note that $\mathcal{A}$ does not have to be free. Recognition is a concept that can be applied to arbitrary algebras and subsets.

If we stay within the interpretation of recognition as a way to represent a language in a compressed way, a natural question comes up that asks how to find the smallest algebra that recognizes a language. In the world of finite words and monoids, this smallest algebra is the syntactic monoid. The syntactic monoid is given through a syntactic congruence. We want such a mechanism for general algebras. Given a language that is a subset of an algebra $\mathcal{A}$, we are interested in the smallest recognizing algebra of the same signature. The recognizing homomorphism in this case should be not be generalized. Similar to the word case we define a syntactic congruence, which generalizes the one for words. Recall that the evaluation of a context is a function.

Definition 25 (Syntactic congruence)**.** *Given is an algebra $\mathcal{A} = (\mathbb{D}; O)$ of S sorts as well as a language $L \subseteq \mathbb{D}_s$ for some $s \in [S]$. For $r \in [S]$, two elements $u, v \in \mathbb{D}_r$ are syntactically congruent if for all contexts $c \in \mathbb{C}^r_s(\sigma)$ holds that*

$$\mathrm{eval}_{\mathcal{A}}(c)(u) \in L \;\Leftrightarrow\; \mathrm{eval}_{\mathcal{A}}(c)(v) \in L.$$

We write $u \sim_L v$.

The congruence $\sim_L$ is precisely the coarsest congruence on $\mathcal{A}$ such that $x \in L$ and $y \notin L$ implies that $x \not\sim_L y$. We call $\mathrm{Synt}(L) = \mathcal{A}/{\sim_L}$ the <u>*syntactic algebra*</u> of L and $\eta_L \colon \mathcal{A} \to \mathcal{A}/{\sim_L}$ we call the <u>*syntactic homomorphism*</u> for which $x \mapsto [x]_{\sim_L}$. This is precisely the natural homomorphism $\pi_{\sim_L}$.

Proposition 26. *Given an algebra $\mathcal{A}$ and a language L in $\mathcal{A}$, then L is recognized by* $\mathrm{Synt}(L)$ *and η_L.*

Proof. Let $X = \eta_L(L)$. Clearly, we have $L \subseteq \eta_L^{-1}(X)$. To show that the statement holds we only need to show $L \supseteq \eta_L^{-1}(X)$. So, suppose there exists $x \notin L$ and $y \in L$ such that $\eta_L(x) = \eta_L(y) \in X$. This implies $x \sim_L y$, which is a contradiction. □

Proposition 27. *Given an algebra $\mathcal{A}$, a language L in $\mathcal{A}$, and an algebra $\mathcal{B}$ that recognizes L, then* $\mathrm{Synt}(L)$ *divides $\mathcal{B}$.*

Proof. Let L be recognized by a homomorphism $\phi \colon \mathcal{A} \to \mathcal{B}$. The algebra $\phi(\mathcal{A}) = \mathcal{B}'$ is a subalgebra of $\mathcal{B}$. To prove that the statement holds we have to construct an epimorphism $\psi \colon \mathcal{B}' \twoheadrightarrow \mathrm{Synt}(L)$. All elements of $\mathcal{B}'$ are of the form $\phi(a)$ for some $a \in \mathcal{A}$. Now, we choose ψ as $\phi(a) \mapsto \eta_L(a)$. This is unambiguous since $\phi(a) = \phi(b)$ implies that $a \sim_L b$ due to recognition, and so $\eta_L(a) = \eta_L(b)$. Surjectivity of ψ also follows. □

Considering the previous proof, note that if ϕ is a non-generalized homomorphism, so is ψ.

The previous proposition tells us that the syntactic algebra is unique and the smallest recognizing algebra. Hence, syntactic algebras are useful objects to define and decide properties of languages.

In the setting of languages of finite words, the notion of regular languages has several equivalent characterizations. The original definition was based on regular expressions. The notion of regularity for the more general framework is based on algebra.

Definition 28 (Regularity). *A language is called regular if its syntactic algebra is finite.*

Thus, whenever we speak of regularity it has to be related to an algebra. For example, a language of well-matched words can be regular with regard to a certain algebra fitted to tree-like structures that we will cover later, however, it is then not necessarily regular with regard to monoids.

4.1 Regular Word Languages

In the word case, as already outlined, the concept of syntactic algebra translates to the classical notion of syntactic monoid. The syntactic congruence is described by the following statement: $x, y \in \Sigma^*$ then $x \sim_L y$ is true if for all $u, v \in \Sigma^*$ holds that $uxv \in L \Leftrightarrow uyv \in L$.

Following our regularity definition, word languages that have a finite syntactic monoid are the regular languages. Regular languages that have an aperiodic syntactic monoid we call *aperiodic*. Regular languages that have a quasiaperiodic syntactic homomorphism we call *quasiaperiodic*. The aperiodic languages coincide with the star-free languages. These are languages we get through star-free expression [Sch65]. General regular expressions yield the regular languages as a whole.

Note that non-regular languages have an infinite syntactic monoid, and so the algebraic framework becomes less useful. For example, even the rather simple language of palindromes has the largest possible syntactic monoid Σ^*. However, there are algebraic approaches to non-regular languages. In the case of non-regular word languages, for example, there is the concept of typed monoids [BKR11]. Another possibility that works for a certain superset of the regular languages will be introduced later.

4.2 Regular Tree Languages

In a first simple special case, we can consider binary trees. Consider the free magma generated by a single element. Terms over such an algebra are just binary trees. The key here is the non-associativity of the operation, which preserves the tree represented by the term. Hence, magmas can be used for recognizing sets of unlabeled binary trees and finite magmas can recognize regular sets of binary trees.

For ranked tree languages in general we need algebras with operations for each letter. Through a ranked alphabet (Σ, r) we define the signature $\sigma = (r(a))_{a \in \Sigma}$. The free algebra of trees over (Σ, r) is $\mathcal{T}(\sigma)$ and we write $\mathcal{T}(\Sigma, r)$. Note that there must exist letters a with $r(a) = 0$ to be assigned to the leaves. This leads to the observation that single-sorted term algebras are the same as the free algebras for ranked tree languages. The notions like recognition and regularity follow from the general definitions.

For unranked trees things become more complicated. The children of a node form a word that consists of trees, or equivalently, nodes have a single child, which is a forest; we will employ the latter view. This forest may or may not be ordered. Notice that we focus on ordered forests and that the unordered case can be embedded via commutativity of the algebra we are about to define.

For a different perspective, consider unranked trees as binary trees: If a node has a number of children, they all can be combined via some binary associative operation. So, to algebraically capture what we described, we need an algebra that has a monoidal, i.e. associative, operation, which allows for concatenating forests. This enables us to assign nodes an unbounded number of child trees.

To derive more natural concepts we actually will consider forest languages instead of tree languages. Moreover, we have to make a design decision: In contrast to the ranked case, a letter does not tell us the number of children of a node, so a node labeled some letter a could be a leaf or an inner node. Inner nodes translate to unary operations in the algebra that take the word of child trees. Leaves could now be modeled through 0-ary operations or we just treat them as inner nodes but have to give them empty forests as children. The first option basically leads to us assigning leaves letters from a separate alphabet. Both versions are ultimately equivalent. The literature is not consistent about which way to use [BW08, BSW12]. We will use the version using empty forests as leaves.

4.2.1 Extend Algebras

There are many reasons why it is desirable to have a recognition mechanism for unranked forest languages. The first one that is presented uses so-called extend algebras. As we already pointed out, a node can have an ordered sequence of children and these children in turn are trees. Hence, we can interpret the children as a forest. This forest is then subject to some unary operation that makes the trees of the forest children of a common root node. We want to call such an operation extend operation. Besides, we need an associative binary operation to compose sequences of forests and a constant for the empty forest. This is in line with the way we denote unranked trees. For example, if a tree has a root labeled a then the children are are sequence of trees that are a forest: $f = t_1 + \ldots + t_n$. The following algebra will incorporate $+$, as well as an operation for $a(\cdot)$, which is one of the mentioned extend operations. Extend algebras have a close connection to forest algebras, which we cover subsequently. After giving the definitions, we will also go into more contextual detail.

Definition 29 (Free extend algebra)**.** *The free extend algebra for unranked forests over an alphabet Σ is defined as the term algebra $\mathcal{T}(2, 1^{|\Sigma|}, 0)/{\sim}$, where $\sim$ is the congruence that makes $+$ associative and 0 neutral. It is denoted as $\mathsf{EA}(\Sigma)$, the domain of this algebra as $H(\Sigma)$, and the operations as $+$,$\triangle_a$ for all $a \in \Sigma$, and 0. Hence,*

$$\mathsf{EA}(\Sigma) = (H(\Sigma); +, (\triangle_a)_{a\in\Sigma}, 0).$$

The monoid $(H(\Sigma); +, 0)$ is free with a neutral element 0 and called horizontal monoid. For $a \in \Sigma$ the operation $\triangle_a$ is called an extend operation.

Note that we defined an unranked labeled forest as a structure $(V; E, (Q_a)_{a\in\Sigma})$. The set of such structures is isomorphic to $H(\Sigma)$. So, a forest language F is a subset of $H(\Sigma)$. This in turn gives us the syntactic congruence for languages and hence a syntactic extend algebra which we denote as as $\mathrm{Synt}(F)$. For its horizontal monoid we write H_F.

Example 30. *Consider some languages and their representations:*

- *The language of binary trees over a unary alphabet $\{a\}$ can be recognized by magmas, as we already indicated. If we want to recognize it with an extend algebra, we do so by a homomorphism $\mathsf{EA}(\{a\}) \to (M; +, \triangle_a, 0)$. The monoid $(M; +, 0)$ with $M = \{0, m, mm, \bot\}$ being commutative and defined by the equation $m^3 = \bot$ for $\bot$ being the absorbing element. Further, $\triangle_a(\bot) = \triangle_a(0) = \triangle_a(m) = \bot$, and $\triangle_a(mm) = 0$.*

- *Consider the forest language F over Σ that consists of forests that only contain trees of size one, omitting the empty forest, which is a child of these single nodes. Such a language resembles a word language $L_F \subseteq \Sigma^*$. When $\mathrm{Synt}(L_F) = (M; +, 0)$ then the syntactic extend algebra of F is $(M; +, E, 0)$ where E are the extend operations. It follows that by using generalized homomorphisms* *we can recognize word languages by extend algebras. This can be achieved by mapping the concatenation of the monoid of the word side to + on the extend algebra side, as well as mapping letters to trees of size one that have the according label. This mapping is almost trivial, however this homomorphism is generalized since the signatures of domain and image differ.*

- *The language that contains forests that consist solely of linear trees is the next example. Here, an element has the form* $(\triangle^*(0))^* = (\triangle^*(0)) + \ldots + (\triangle^*(0))$. *Note that we used the Kleene star* $*$ *for different operations. The inner star in* $(\triangle^*(0))^*$ *indicates an arbitrary number of applications of* $\triangle$ *operations whereas the outer star refers to the* $+$ *operation. So, through the subexpression* $\triangle^*(0) = \triangle(\triangle(\ldots(\triangle(0))))$ *we get linear trees.*

- *Generalized homomorphisms are useful to express certain mappings. For example, consider the language* $L = \{\triangle_a, \triangle_b\}^*(0)$. *It consists of vertical lists arbitrarily labeled by* a *and* b. *Now, let* $\phi\colon \mathsf{EA}(\Sigma) \to \mathsf{EA}(\Sigma)$ *be a homomorphism with* $\triangle_a \mapsto \triangle_a$ *and* $\triangle_b \mapsto t$ *where* t *is a context with* $t(x) = \triangle_b(x) + \triangle_a(0)$. *It equips every node labeled* b *a sibling labeled* a.

We saw examples using generalized homomorphisms. They are indeed required for many natural mappings. For example, if we want to map forests over some alphabet into a different alphabet, this already needs generalized homomorphisms when using extend algebras, since letters come into being through unary operations and not 0-ary ones. In fact, non-generalized homomorphisms can only alter the leaves of the forest.

As we saw, generalized homomorphisms can also be used to recognize forest languages using algebras of a different signature. For example, the set of all forests can be recognized by the trivial monoid. However, allowing for generalized homomorphisms does not mess up our regularity definition. If an unranked tree language is recognized by a finite algebra via some generalized homomorphism, then its syntactic monoid is finite.

4.2.2 Forest Algebras

Extend algebras are designed for unranked forests. There exists another class of algebras for exactly that purpose, which are the forest algebras [BW08]. Forest algebras are closely related to extend algebras. A forest algebra has two domains H and V. The domain H is the same as the domain in the extend algebra and V is the set generated by the extend operations. Both form a monoid. The monoid H corresponds to forests, which we can concatenate. In V we have contexts instead. The operation concatenates contexts vertically, i.e. the vertical concatenation of two contexts is obtained by replacing the variable in the context with the other context. This way V can be considered a subset of H^H. The monoid H is called the horizontal monoid and V the vertical monoid. The benefit of having the vertical monoid present is that we can try to retrieve properties from it that are not directly apparent in the extend operations.

Definition 31 (Free forest algebra)**.** *The free forest algebra for unranked forests over an alphabet Σ is defined as the two-sorted term algebra*

$$\mathcal{T}((11,1),(12,2),(21,2),(22,2),(21,1),(2)^{|\Sigma|},(1),(2))/\!\sim,$$

which we denote as $\mathsf{FA}(\Sigma)$, which in turn is denoted as

$$(H(\Sigma),V(\Sigma);+,+',+'',\cdot,\cdot',(\triangle_a)_{a\in\Sigma},0,1).$$

The relation $\sim$ is the congruence that resembles the following equations, where $h,h_1,h_2\in H(\Sigma)$ and $v,v_1,v_2\in V(\Sigma)$:

- *The operation $+$ is associative and 0 is neutral, i.e. $(H(\Sigma);+,0)$ is a monoid.*
- *The operation $\cdot$ is associative and 1 is neutral, i.e. $(V(\Sigma),\cdot,1)$ is a monoid.*
- $(h_1+h_2)+'v=h_1+'(h_2+'v)$
- $(v+''h_1)+''h_2=v+''(h_1+h_2)$
- $(h_1+'v)+''h_2=h_1+'(v+''h_2)$
- $(v_1\cdot v_2)\cdot'h=v_1\cdot'(v_2\cdot'h)$

The monoid $(H(\Sigma),+,0)$ is called the horizontal monoid and $(V(\Sigma),\cdot,1)$ the vertical monoid.

The operation $+$ concatenates two forests, while $+'$ and $+''$ concatenate a forest and a context, which results in a context. The operation $\cdot$ concatenates contexts vertically and $\cdot'$ inserts a forest into the hole of a context resulting into a forest. From now on we will not distinguish between $+$, $+'$, and $+''$ when writing down a forest. Furthermore, we will drop $\cdot$ and $\cdot'$ in the notation. Through associativity we can also drop some parentheses. So, for example, we may write $v_1(h_1+h_2)+v_2\in V(\Sigma)$ and even $\mathsf{FA}(\Sigma)=(H(\Sigma),V(\Sigma);+,\cdot,(\triangle_a)_{a\in\Sigma},0,1)$. Also, pay attention to the extend operations $\triangle_a$. For all practical purposes they are the same as in extend algebras, but formally in forest algebras they are not unary operations over the horizontal domain but constants from the vertical domain. So, technically we cannot write $\triangle_a(h)$ for $h\in H(\Sigma)$. However, we still do so and keep in mind that the correct way for writing this would be $\triangle_a\cdot'h$.

Note that a few design choices were made. The operation $\cdot'$ is considered an *action* in the original paper. Also, they do not have $+'$ and $+''$ there, but rather have two operations, each taking a forest and append a hole to the left, or to the right respectively. As already mentioned, whether or not leaves have a separate alphabet is also an issue. In the different papers [BW08] and [BSW12] we find different versions. The properties we packed into $\sim$ can be found in the original definitions in a different presentation.

It is not a coincidence that the domain of the extend algebra has the same name as the first subdomain of the forest algebra. As we will see, the horizontal monoid is actually identical for extend and forest algebras, so both contain labeled finite unranked forests of the form $(V; E, (Q_a)_{a\in\Sigma})$. In the case of forest algebras, we additionally have the vertical monoid consisting of contexts. However, a forest language F still is just a subset of the horizontal monoid.

That way we get the syntactic congruence for languages and hence a *syntactic forest algebra*. We write $\mathrm{Synt}(F)$ and it is clear from the context whether we mean forest or extend algebras. In correspondence to extend algebras the horizontal monoid is denoted by H_F. The vertical monoid is denoted by V_F.

For an alphabet Σ there exists a meaningful bijection between the horizontal monoids of $\mathsf{EA}(\Sigma)$ and $\mathsf{FA}(\Sigma)$, which we could regard as an isomorphism. First, every term of the free algebra $\mathsf{EA}(\Sigma)$ can be found again in $\mathsf{FA}(\Sigma)$. We see this by induction over terms. The operation $+$ translates to $+$ again. In the case of the unary operation $\triangle_a$ we have a term of the form $\triangle_a(f)$. This translates to $\triangle_a \cdot' f'$, where now $\triangle_a$ is a constant operation and f' is the result of the translation of f, which we get by induction. Finally, the constant 0 stays 0. For the converse, notice that $\mathsf{FA}(\Sigma)$ possesses several operations: $+$, $+'$, $+''$, $\cdot$, $\cdot'$, $\triangle_a$, 0, and 1. Terms can have quite different shapes while still being equivalent in the free forest algebra due to the properties we enforced in the definition, but these enable us to convert terms into a form, which is close to the corresponding term for the extend algebra. To do so we get rid of $+'$, $+''$, $\cdot$, and 1. Whenever a term contains one of the mentioned binary operations, the result is a context, which has to be filled in later. This delayed filling-in we now remove. Consider a term t, which has a subterm $f +' c$. Now, observe that this subterm is a context again and the forest we insert into the context can be found as a different subterm in t. Let t' be this subterm. Now, c has necessarily at least one leaf labeled by 1 and exactly one of these is the place where t' gets inserted. Thus, we replace this leaf and insert t' at this place directly. Since now certain subterms turn from context to forest, we may have to change $+'$ or $+''$ to $+$, or $\cdot$ to $\cdot'$. The original occurrence of t' is deleted as well as an operation $\cdot'$, which was responsible for inserting t'. We now eliminated $+'$ and perform a similar procedure for $+''$ and $\cdot$ and repeat until all are eliminated. The result then also does not have leaves labeled 1 any more. The procedure can also be seen as a reparenthesising procedure, which maintains correct operation symbols. For example, for $h \in H(\Sigma)$ the forest $((h +' \triangle_a) \cdot \triangle_a) \cdot' 0$ can be also written as $h + (\triangle_a \cdot' (\triangle_a \cdot' 0))$.

This construction shows that the horizontal monoids of free extend algebras $\mathsf{EA}(\Sigma)$ and free forest algebras $\mathsf{FA}(\Sigma)$ indeed coincide. This also translates to quotients of algebras. Also, note that the horizontal monoid of a free forest algebra $\mathsf{FA}(\Sigma)$, in fact, is of the form $\mathbb{T}(\sigma)$ and the vertical monoid of the form $\mathbb{C}(\sigma)$ for an appropriate signature σ.

Lemma 32. *For an alphabet Σ the free extend algebra $\mathsf{EA}(\Sigma)$ and the free forest algebra $\mathsf{FA}(\Sigma)$ have the same horizontal monoid $H(\Sigma)$. A congruence $\sim$ on $H(\Sigma)$ in $\mathsf{EA}(\Sigma)$ is also a congruence in $\mathsf{FA}(\Sigma)$ and vice versa, hence the algebras $\mathsf{EA}(\Sigma)/\sim$ and $\mathsf{FA}(\Sigma)/\sim$ have the same horizontal monoid.*

Proof. We already argued that $\mathsf{EA}(\Sigma)$ and $\mathsf{FA}(\Sigma)$ have the same horizontal monoid $H(\Sigma)$. Let now $\sim$ be an equivalence relation on $H(\Sigma)$. If it is a congruence in $\mathsf{EA}(\Sigma)$, then it is also one in $\mathsf{FA}(\Sigma)$: For $u, v, w, x \in H(\Sigma)$ we have $u + v \sim w + x$ if $u \sim w$ and $v \sim x$, which translates directly to $\mathsf{FA}(\Sigma)$. Also, $\triangle_a(u) \sim \triangle_a(v)$ if $u \sim v$, which translates to $\triangle_a(1) \cdot' u \sim \triangle_a(1) \cdot' v$ if $u \sim v$. The converse follows similarly.

□

The lemma tells us that both free extend and free forest algebras contain the same structures and also that the syntactic objects are equivalent. Actually, for any given extend algebra there exists a unique *corresponding* forest algebra and vice versa. The corresponding forest algebras have the same horizontal monoid and the unary operations $\triangle_a$ of the extend algebra coincide with the constant operations $\triangle_a$ in the forest algebra. In this case there is an isomorphism between both.

The previous considerations underline the fact that the difference between an extend and a forest algebra can be interpreted as a precomputation. A forest algebra contains information about the vertical behavior more explicitly than the extend algebra, as $V(\Sigma)$ is just the set of contexts we can generate through the $\triangle_a$ operations.

In [BW08] the forest algebra framework was laid out, which also included homomorphisms. The authors defined a forest algebra homomorphism to actually consist of two homomorphisms: One for the horizontal and one for the vertical monoid. This notion coincides with our notion of non-generalized homomorphisms.

Now consider the case of homomorphisms for extend and forest algebras. In a forest algebra, a non-generalized homomorphism may map each $\triangle_a$ to an arbitrary element of the vertical monoid in the target forest algebra. In an extend algebra, however, $\triangle_a$ is a unary operation. If we wanted to achieve the equivalent as in the forest algebra case, we need a homomorphism, which assigns $\triangle_a$ a context, but as this is a generalized homomorphism, it means that forest algebra homomorphisms translate to generalized homomorphisms between extend algebras.

For the following proposition recall that we defined a homomorphism to be a mapping from the domain of one algebra to the domain of another. Whether it is generalized or not depends on how this mapping can be achieved using α, which assigns terms terms to operations. So, if we have forest algebras F_1 and F_2 and corresponding extend algebras E_1 and E_2, then a homomorphism ϕ from F_1 to F_2 maps forests as well as contexts. If we restrict ϕ on forests, we get a map from E_1 and E_2. The following proposition now shows that this map indeed is a generalized homomorphism.

Proposition 33. *Let F_1 and F_2 be forest algebras, E_1 and E_2 be the corresponding extend algebras, and H be the horizontal monoid of F_1 and E_1. For a non-generalized homomorphism ϕ between F_1 and F_2, there exists a generalized homomorphism ϕ between E_1 and E_2 with $\phi(h) = \psi(h)$ for $h \in H$.*

Proof. Suppose that F_1 and F_2 have the same signature σ. The signature could only differ because of different alphabet sizes, so we assume the alphabets to be equal.

We know that F_1 and E_1 as well as F_2 and E_2 have the same horizontal monoids. Based on this fact we may assume that any term for F_1 is also isomorphic to a term for E_1, i.e. it uses only $+$ and $\cdot'$ as binary operations and $\triangle_a$ as constants. Since ϕ is not generalized, $+$ and $\cdot'$ get mapped onto $+$ and $\cdot'$ in F_2. The constants $\triangle_a$, however, get replaced by a context c. We can translate this now into a generalized homomorphism between E_1 and E_2 that realizes the same map between the horizontal monoids. Here, $+$ is again mapped onto $+$, but the unary operation $\triangle_a$ of E_1 gets mapped onto c, which is a term with a variable, which has the same signature as $\triangle_a$. This is a generalized homomorphism and the mapping realized is preserved.

□

Example 34. *Some simple cases for properties of forest algebras are the following:*

- *Given a forest language F, we can ask complexity questions. For example: Has the language a* $\mathsf{FO}[<]$ *formula? This is a formula using first-order quantification and an ancestor predicate $<$. By the connection between aperiodic monoids and logic we know from the word case, it is easy to see that a necessary condition for F to be in* $\mathsf{FO}[<]$ *is that the horizontal and the vertical monoid of the syntactic forest algebra need to be aperiodic.*
- *Again, given a forest language F, we can ask whether it is regular in the word sense, which means asking whether* $\mathsf{wm}(F)$ *is a regular language. Using a pumping argument one can see that, if F contains arbitrarily deep trees,* $\mathsf{wm}(F)$

is not regular. On the other hand, if F has a bound on the tree depth, $\mathsf{wm}(F)$ *is regular, since we can count to a constant in the states of a finite automaton. Bounded depth can be decided using the syntactic forest algebra. First, V has to be nilpotent. Further, let $\perp$ be the null element of V, then the sufficient condition is that $\perp(0)$ is not in the accepting set.*

- *If we continue the previous example and check for nilpotency, but this time in H instead of V, we get the property that captures bounded rank of nodes in the trees.*
- *We say that a forest algebra is distributive if $c(h_1 + h_2) = ch_1 + ch_2$ holds for $c \in V$ and $h_1, h_2 \in H$. A forest algebra that is distributive has the property that it cannot distinguish between forests that have the same path language, whereas the path language of a forest is the set of words that can be read from roots to leaves. Actually, this set is ordered. If H is commutative, this order no longer exists.*
- *The yield of a forest is the word language that is obtained by an in-order traversal of all leaves. Yields of regular tree languages are precisely the context-free word languages. One can observe now that, if H is commutative, so is the yield language.*

4.3 Regular Languages of Nested and Well-Matched Words

In the second chapter we discussed the nested and well-matched word structures, and saw how closely related they are to each other and to unranked labeled forests. The goal of this section is to define algebras for nested and well-matched word languages. To that end we will use what we have established for unranked forest languages.

First, however, we have to take care of an issue: Well-matched words may have internal letters. Equivalently, nested words may have positions that are not part of a matching. Both do not have a natural resemblance in unranked trees. There are different ways to evade this problem. One option is just allowing internal letters to appear in the corresponding forest, but then it must be ensured that only leaves are labeled with these letters. This can be troublesome when we want to come up with a free algebra. To solve that, one could introduce new 0-ary operations to the algebra, which correspond to the internal letters. We could also just disallow internal letters and simulate them by a pair of call and return letters. This mapping, however, changes the length of the word and it is not a length-multiplying homomorphism, which may cause problems in some areas.

To begin with, we execute the following under the assumption that there are no internal letters present, or, equivalently, that all positions are matched.

Given a nested word w over Σ, $\mathsf{forest}(w)$ gives us the corresponding forest over Σ^2. We use Σ^2 since a pair of letters that is in matching positions ends up in a single node. Now, the free algebras $\mathsf{EA}(\Sigma^2)$ and $\mathsf{FA}(\Sigma^2)$ can be utilized as the free algebras for nested words over Σ. If L is a nested word language, $\mathsf{forest}(L)$ gives us the corresponding forest language and the syntactic extend or forest algebra of $\mathsf{forest}(L)$ is now also the syntactic algebra of L. We may regard the horizontal monoid of the free algebra as a set of nested words. The vertical monoid of the free forest algebra contains all contexts and in the case of nested words it consists of nested words that have some marked position, telling that after this position another nested word may be inserted. Two contexts can be concatenated by inserting one into the hole of another. The horizontal monoid of the syntactic forest algebra of language of nested words L is $H_{\mathsf{forest}(L)}$, but for simplicity we just write H_L. Similarly, we write V_L.

Well-matched words can be treated similarly. If $\hat{\Sigma} = (\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \emptyset)$ is the alphabet for the well-matched words, we get forests over $\Sigma_{\text{call}} \times \Sigma_{\text{ret}}$. In this case, a context is a pair of words (u, v) such that uv is well-matched. For a language L of well-matched words we write H_L and V_L to address its horizontal and vertical monoids.

Now back to the issue of internal letters. The algebra $\mathsf{EA}(\Sigma_{\text{call}} \times \Sigma_{\text{ret}}) = (H; +, (\triangle_a)_{a\in\Sigma_{\text{call}}\times\Sigma_{\text{ret}}}, 0)$ is the free extend algebra for well-matched words over $\hat{\Sigma} = (\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \emptyset)$. For $\hat{\Sigma} = (\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \Sigma_{\text{int}})$ we can define $(H; +, (\triangle_a)_{a\in\Sigma_{\text{call}}\times\Sigma_{\text{ret}}}, 0, (\diamond_a)_{a\in\Sigma_{\text{int}}})$, where $\diamond_a$ is a 0-ary operation for all $a \in \Sigma_{\text{int}}$. In the case of forest algebras we can do the same. Hence, we define $\mathsf{EA}(\hat{\Sigma})$ to be the modified extend algebra we just constructed and $\mathsf{FA}(\hat{\Sigma})$ to be the corresponding modified forest algebra. That means that $\text{WM}(\hat{\Sigma})$ can be considered the horizontal monoid of the free forest algebra. All theory we developed for extend and forest algebras follows immediately.

A language of nested words is called *regular* if its syntactic extend algebra is finite. A language of well-matched words is also called regular if its syntactic extend algebra is finite. In this case we call it a *visibly pushdown language* (VPL) based on visibly pushdown automata, which we will cover later.

In [AKMV05] a congruence $\equiv_L$ for well-matched words that characterizes the VPLs has been investigated. It is basically a modified syntactic congruence. For a language of well-matched words L, the well-matched words x and y are in relation $x \equiv_L y$ if and only if for all words $u, v \in \Sigma^*$ holds that $uxv \in L \Leftrightarrow uyv \in L$. The paper states that $\equiv_L$ has finite index if and only if L is a VPL. What the paper omits to say is that $\text{WM}(\hat{\Sigma})/{\equiv_L}$ forms a monoid with concatenation and $[\epsilon]_{\equiv_L}$ as neutral element. However, this monoid is not a recognizing object, of course, since finite monoids can only recognize regular word languages. Nevertheless, it turns out that this monoid is precisely the horizontal monoid of the syntactic forest algebra. To see that, consider $u, v \in \Sigma^*$ as used in the definition of $\equiv_L$. If we add the requirement that $uv \in \mathrm{WM}(\hat{\Sigma})$, the congruence does not change. So, (u, v) is a context and $\equiv_L$ coincides with $\sim_L$.

As mentioned before, typed monoids are an approach to capture non-regular word languages algebraically. We do not know whether VPLs in general can be captured by typed monoids, but for certain subsets this might be possible.

4.4 Conclusion

Summary

Initially, we defined what languages are. There are different kinds of languages; we considered languages of words, trees, and well-matched words. In that, languages are subsets of algebras. In this chapter we looked into representing languages by algebras that might be smaller than the original algebra of which the language is a subset of. This representation is called recognition. In the case of words the recognition framework is well-established and classically uses monoids. Languages that can be recognized by a finite algebra we call *regular*. To obtain regularity notions for forest languages we introduced algebras that fit this case. First, we considered extend algebras. An extend algebra is similar to a monoid that is augmented with a number of unary operations that we call *extend operations*. After we established the whole recognition framework we related extend algebras to forest algebras. Forest algebras can be obtained from extend algebras. They have an additional domain, which captures the closure of the extend operations. This domain is called the vertical monoid while the first domain is common to the domain of the corresponding extend algebra, which we call the *horizontal monoid*. This shows that both objects are in a one-to-one correspondence. We also related the homomorphisms accordingly: Forest algebra homomorphisms translate to generalized homomorphisms for extend algebras.

Extend and forest algebras can be used for forest language recognition but also for languages of nested and well-matched words.

Contributions

We introduced recognition is a very general sense in which known recognition schemes embed. Our framework entails syntactic congruences, homomorphisms and syntactic algebras themselves. In our treatment of forest language recognition we introduced forest algebras as emerging from extend algebras. Extend algebras are a new concept. Especially the correspondence for forest algebra homomorphisms and generalized homomorphisms on the extend algebra side underpins the utility of the notion of generalized homomorphisms.

Also, we connected the algebra framework for forest languages to languages of well-matched words, for which only the insufficient congruence by Alur et al. was known [AKMV05].

Sources and Related Work

The recognition framework for words is ubiquitous formal language theory. Literature is split in those contributions working with semigroups and those working with monoids. We chose to join the monoid side. Among the many books and papers surveying the topic we refer to [HU79, Str94].

The algebraic treatment of forest languages is much younger. Here, we relied on the work of Bojańczyk and Walukiewicz [BW08] which introduced forest algebras. There are also other attempts for an algebraic treatment. For example, Alur et al. [AKMV05] showed a congruence for VPLs. This congruence constitutes a monoid which, as it turns out, coincides with the horizontal monoid. However, this object is too weak for recognition purposes.

Examples for the application of forest algebras can be found in [BW08, BSW12, Str13, KS15].

Further Research

Forest algebras are already in use and proved to be useful. Hence, there should be potential for extend algebras as well. It depends on the application whether extend or forest algebras are more handy. One example where extend algebras could be preferred is the modern equational approach to language and complexity theoretic questions like in [CK16, BCGK17, GKP14].

Chapter 5

Logic

After algebra, logic is the second recognition mechanism we consider. Logic formulas typically operate on structures instead of elements of a free algebra. The study of logic in combination with different kinds of structures is called model theory. In logic we build formulas that may utilize relations given in the structure. Further, there is quantification over the domain of the structure. By applying the semantics of the formulas we can decide whether some structure S is a model for some formula ϕ, which we denote as $S \models \phi$. When we relate formulas and structures, it is always clear from the context what kind of structures we address, indicating whether we consider words trees, etc. The set of all structures that satisfy a formula ϕ is denoted as $L(\phi) = \{S \mid S \models \phi\}$, which is then a set of words, trees, etc. Therefore, $S \models \phi$ if and only if $S \in L(\phi)$.

For $S \models \phi$ to hold it is necessary that both S and ϕ follow the same format, which means they have the same signature. It is straightforward to define a logic framework for the many-sorted case, but we refrain from that as we would not make use of it. So, signatures are elements of $\mathbb{N}^*$ that only assign arities to relations.

Definition 35 (Monadic second-order formula (MSO))**.** *Given a signature $\sigma \in \mathbb{N}^k$ for $k \in \mathbb{N}$, let $\mathcal{V}_1$ and $\mathcal{V}_2$ be finite sets, where $\mathcal{V}_1$ contains first-order variables and $\mathcal{V}_2$ second-order variables. Then an MSO formula is defined as follows:*

- *$R_i(x_1, \dots, x_{\mathsf{Ar}_\sigma(i))})$ is an MSO formula with free variables $\mathcal{V}_1 = \{x_1, \dots, x_{\mathsf{Ar}_\sigma(i))}\}$ and $\mathcal{V}_2 = \emptyset$.*
- *$X(x)$ is an MSO formula with free variables $\mathcal{V}_1 = \{x\}$ and $\mathcal{V}_2 = \{X\}$.*
- *If ϕ is an MSO formula with free variables $\mathcal{V}_1$ and $\mathcal{V}_2$, then $\neg\phi$ is an MSO formula with free variables $\mathcal{V}_1$ and $\mathcal{V}_2$.*

- *If ϕ is an MSO formula with free variables $\mathcal{V}_1$ and $\mathcal{V}_2$ and ψ is an MSO formula with free variables $\mathcal{V}_1'$ and $\mathcal{V}_2'$, then $\phi \wedge \psi$ as well as $\phi \vee \psi$ are MSO formulas with free variables $\mathcal{V}_1 \cup \mathcal{V}_1'$ and $\mathcal{V}_2 \cup \mathcal{V}_2'$.*

- *If ϕ is an MSO formula with free variables $\mathcal{V}_1$ and $\mathcal{V}_2$, then $\exists x\phi$ as well as $\forall x\phi$ are MSO formulas with free variables $\mathcal{V}_1 \setminus \{x\}$ and $\mathcal{V}_2$.*

- *If ϕ is an MSO formula with free variables $\mathcal{V}_1$ and $\mathcal{V}_2$, then $\exists X\phi$ as well as $\forall X\phi$ are MSO formulas with free variables $\mathcal{V}_1$ and $\mathcal{V}_2 \setminus \{X\}$.*

- *Formulas with $\mathcal{V}_1 = \emptyset$ and $\mathcal{V}_2 = \emptyset$ are called closed.*

Small variable letters indicate first-order variables. Capitalized variable letters indicate monadic second-order variables.

Note that predicates are the atomic formulas. We do not write all predicates exactly the way they were given in the definition. For example, for the comparison predicate, we like to write as $x < y$ instead of $<(x, y)$.

To define the semantics we fix some structure $S = (\mathbb{D}; O)$ that has the same signature as an MSO formula we define the semantic for. Let now $\nu_1 \colon \mathcal{V}_1 \to \mathbb{D}$ be a valuation of the variable set $\mathcal{V}_1$ and let $\nu_2 \colon \mathcal{V}_2 \to 2^{\mathbb{D}}$ be a valuation for the variable set $\mathcal{V}_2$. By $\nu_1^{x\mapsto d}$ we denote the valuation we get if in ν_1 the value $\nu_1(x)$ is set to d, $\nu_2^{X\mapsto d}$ is defined similarly. Now, if ϕ and ϕ' are formulas with free variable sets $\mathcal{V}_1$ and $\mathcal{V}_2$ and V_1' and V_2' respectively, we define the semantic for the different cases:

- $S \models^{\nu_1,\nu_2} R_i(x_1, \ldots, x_{\mathsf{Ar}_\sigma(i)})$ if $(\nu_1(x_1), \ldots, \nu_1(x_{\mathsf{Ar}_\sigma(i)})) \in R_i$ for $R_i \in O$ being the i-th relation of S.

- $S \models^{\nu_1,\nu_2} X(x)$ if $\nu_1(x) \in \nu_2(X)$.

- $S \models^{\nu_1,\nu_2} \neg\phi$ if $S \not\models^{\nu_1,\nu_2} \phi$

- $S \models^{\nu_1\cup\nu_1',\nu_2\cup\nu_2'} \phi \wedge \phi'$ if $S \models^{\nu_1,\nu_2} \phi$ and $S \models^{\nu_1',\nu_2'} \phi'$.

- $S \models^{\nu_1,\nu_2} \exists x\phi$ if there exists some $d \in \mathbb{D}$ such that $S \models^{\nu_1^{x\mapsto d},\nu_2} \phi$.

- $S \models^{\nu_1,\nu_2} \exists X\phi$ if there exists some $d \in 2^{\mathbb{D}}$ such that $S \models^{\nu_1,\nu_2^{X\mapsto d}} \phi$.

- If ϕ it is closed and $S \models^{\emptyset,\emptyset} \phi$, we write $S \models \phi$.

In the word case we can, for example, have structures of the form $([n]; <, (Q_a)_{a\in\Sigma})$. This one has a signature of $(2, 1^{|\Sigma|})$. Every formula of the same signature fits such a word. Also, when writing a formula down, for readability, we directly use the predicate names from the structure like $x < y$ or $Q_a(x)$.

Besides existential and all quantification there also exist other quantifiers like modulo and majority quantifiers. Introducing a first-order quantifier MOD_k for $k \in \mathbb{N}$, we assign it the following semantic, which makes the quantification true if there are a multiple of k many valuations.

- $S \models^{\nu_1,\nu_2} \mathrm{MOD}_k x\psi$ if $|\{d \in \mathbb{D} \mid S \models^{\nu_1^{x\mapsto d},\nu_2} \psi\}| \equiv 0 \pmod{k}$.

A first-order quantifier MAJ for majority is satisfied if the majority of all valuations of the variable satisfy.

- $S \models^{\nu_1,\nu_2} \mathrm{MAJ}x\psi$ if

$$|\{d \in \mathbb{D} \mid S \models^{\nu_1^{x\mapsto d},\nu_2} \psi\}| > |\{d \in \mathbb{D} \mid S \not\models^{\nu_1^{x\mapsto d},\nu_2} \psi\}|.$$

The set of first-order definable structures we denote as FO and the MSO definable ones as MSO. It is also common to note the predicates used, e.g. $\mathsf{FO}[<]$, however, it is always assumed that the Q_a predicates are accessible without mentioning them explicitly. The type of quantification we allow is, for example, denoted as $\mathsf{FO}+\mathsf{MOD}[<]$ or $\mathsf{MAJ}[<]$. The established notation conventions, however, are not always consistent, so we will later define the logic classes individually.

Substitution is a useful tool in logic. If we define a formula ϕ with two free first-order variables, we may use ϕ in other formulas as if it was a predicate. For example, consider the formula $\neg y < x$. It has two free variables and we may use it as a predicate $\leq$. Substitutions can be also seen as a kind of reduction or transduction. For example, we may have a procedure where as a first step the input is preprocessed. An example, as we will see, lies in computing the matching predicate for well-matched words. This may lead to an output that is the input enhanced by some additional information in the form of a larger alphabet. Now, the formula for the second step may access this larger alphabet and the querying predicates Q_a actually have to be replaced by formulas for the first step.

5.1 Logic on Words

A word is a structure of the form $([n]; <, (Q_a)_{a\in\Sigma})$, however, it may also have additional predicates, especially numerical predicates like the binary successor predicate $+1$ and a ternary $+$ predicate, which is used in the form $x + y = z$. In the word case $\mathsf{MSO}[+1]$ equals $\mathsf{MSO}[<]$. In a very strict sense, this statement is syntactically incorrect since both sets contain different kinds of structures. However, one can define the $<$ predicate in $\mathsf{MSO}[+1]$. That means that if we have a language

$L \in \mathsf{MSO}[<]$ of words $([n]; <, (Q_a)_{a\in\Sigma})$, we can substitute $<$ by an MSO construction. So, the equivalent words $([n]; +1, (Q_a)_{a\in\Sigma})$ form a language L', which then is in $\mathsf{MSO}[+1]$. That is why we simply speak of MSO.

The MSO definable languages are precisely the regular languages [Bü60].

The set $\mathsf{FO}[\mathsf{arb}]$ consists of all languages that we get through arbitrary numerical predicates. The set $\mathsf{FO}[\mathsf{Reg}] = \mathsf{FO}[<, \equiv]$ equals $\mathsf{MSO}[+1] \cap \mathsf{FO}[\mathsf{arb}]$. So, $\mathsf{FO}[\mathsf{Reg}]$ captures exactly the regular languages in $\mathsf{FO}[\mathsf{arb}]$. This is also characterized by quasiaperiodicity of the syntactic homomorphism [BCST92].

Another subset is $\mathsf{FO}[<]$, which corresponds to aperiodic syntactic monoids and star-free expressions [MP71, Sch65].

5.2 Logic on Trees

In the case of trees one has to distinguish the different kinds. We will not cover the ranked case. In the unranked case we defined regularity as those forest languages that are recognized by finite forest algebras. So, here we have an ancestor relation, or equivalently, a vertical order and a horizontal order. Note that an ancestor relation is equivalent to the transitive closure of the edge set of the underlying graph. Like in the word case where we can simulate $<$ by $+1$ in the MSO case, the same construction can be used to simulate the ancestor relation by the edge relation. Now, by MSO formulas over labeled unranked ordered forests we get the regular sets [TW68, Don70].

A special case is the unordered one. If we consider labeled unranked forest languages accepted by finite forest algebras with commutative horizontal monoid, we get languages that are captured by MSO formulas that only utilize an ancestor relation.

As a natural restriction the first-order fragments are of great interest. In the word case we have a decidable characterization in terms of aperiodicity and quasiaperiodicity. In the tree case we are missing that and instead have a major open problem here.

5.3 Logic on Graphs

In graphs we are mostly interested in MSO definable sets due to the Theorem of Courcelle [Cou90].

Theorem 36 (Courcelle). *Let $\mathcal{G}_k$ be the set of graphs of tree-width k for some $k \in \mathbb{N}$ and let ϕ be some MSO formula. Then it is decidable in linear time whether for some $G \in \mathcal{G}_k$ it holds that $G \models \phi$.*

In [EJT10] this result has been improved to logarithmic space. We will come back to Courcelle's Theorem later. This theorem is considered to be very important as many graph problems can be expressed by an MSO formula.

5.4 Logic on Nested Words

A nested word has the form $([n]; <, (Q_a)_{a\in\Sigma}, \rightsquigarrow)$ where $\rightsquigarrow$ is a nesting relation. MSO formulas over nested words are like formulas over ordinary words but may use $\rightsquigarrow$ also. Of course, the languages recognized by such MSO formulas are the regular ones with respect to nested words.

The difficulty of finding a decidability result for first-order definability is inherited from the unranked tree case.

In the well-matched word case we have just ordinary words of the form $([n]; <, (Q_a)_{a\in\Sigma})$. If we want to capture VPLs by MSO, we have to add $\rightsquigarrow$ as a matching relation [AM04]. To decide which VPLs are first-order definable is a subject of chapter 11 where an important issue is how to define $\rightsquigarrow$ in first-order logic.

While the logic for nested words involves a matching predicate, which is part of the nested word itself, there exists a relationship to context-free languages. If we relax the fixed matching predicate, and instead allow it to be existentially quantified within the formula, we get the context-free languages. Formally, for every context-free language L there exists a formula $\exists \rightsquigarrow \phi$ that models L, where ϕ is a first-order formula using $\rightsquigarrow$ [LST94]. This result fits the observation that the context-free languages are yields of regular tree languages. The yield throws away the tree structure and if one wants to derive a formula for such a language, the tree structure has to be guessed, which is what the existential quantification over the matching predicate is doing.

5.5 Conclusion

Summary

We introduced MSO logic for arbitrary signatures and looked at the set of inputs we are interested in, that is words, trees, and nested words.

Contributions

This chapter only serves to provide notation and state some classical results.

Sources and Related Work

Surveys on the topic include [Str94, CDG$^+$07, Tho97, EF95].

Further Research

An obvious mission for the future would be characterizing logic fragments using the algebraic objects defined in the previous chapter.

Chapter 6

Automata

Automata come in different shapes. Finite automata and Turing machines are classical examples for automata that read finite words. The output is then one bit most of the time, telling whether the input is accepted or rejected. However, it is also possible to consider automata that output more information.

Recall that there are the interpretations of inputs as either being structures, or as elements of a free algebra otherwise. In the context of automata we use the structure view although they are also very close to algebra. Automata can not only be built around words but also around other structures like trees. Infinite input structures are also possible but not covered here.

In general, an automaton works by generating a run on the input. Then there is either a condition identifying accepting runs or, more generally, a procedure to compute the output value from the runs.

For each input model, one can examine the corresponding finite automaton model. In this context *finite* means that the storage is finite and implemented by the states. These automata models usually correspond to regular language classes. If we equip automata with additional storage like counters, stacks, or tapes, we get larger classes. Here, we have also to pay attention to the input structure. For example, the set of regular nested word languages is accepted by finite nested word automata. On the other hand, if we consider the isomorphic regular languages of well-matched words, a finite word automaton is too weak; we need a pushdown automaton model for this class of languages.

In this chapter we will look at word-, tree-, and nested word automata. In most cases we will, for the most part, focus on finite automata, with the most notable exception of visibly pushdown automata. We will regard accepting automata and automata that compute more information than the acceptance bit. In the case of a single bit output, we say the automaton *recognizes* or *accepts* a language; in the more general case we say it *implements* a function.

Automata can be used to define complexity classes by restricting resources like time and space. Further, we can use them to get formal languages classes, which is done by restricting their functionality, which leads to the Chomsky hierarchy. Grammars are also closely related, but we will omit introducing them formally.

6.1 Word Automata

6.1.1 Finite Automata

Finite word automata belong to the most basic automata models. They have only constant storage, which is given through the states and no additional storage mechanism. A finite automaton steps through the input word from start to end and changes its state in each step accordingly. If a so-called final state is reached after the word is read, the word is accepted. Finite automata can be deterministic (DFA) and non-deterministic (NFA).

Definition 37 (Finite automaton (on words)). *A (non-deterministic) finite automaton $\mathcal{M}$ is a tuple $(Q, I, F, \Sigma, \delta)$ where:*

- *Q is the finite set of states.*
- *$I \subseteq Q$ is the set of initial states.*
- *$F \subseteq Q$ is the set of final states.*
- *Σ is the alphabet.*
- *$\delta \subseteq Q \times \Sigma \times Q$ is the transition relation.*

An automaton is called deterministic if $|I| = 1$ and the relation δ is a total function $Q \times \Sigma \to Q$.

In the deterministic case we usually write q_I for the initial state instead of $\{q_I\}$. Moreover, in the non-deterministic case we may consider the transition relation δ to be a function $Q \times \Sigma \to 2^Q$.

Given an automaton $\mathcal{M}$ and a word $w \in \Sigma$, $\mathcal{M}$ induces a set of <u>runs</u> on w. A run is a word $\rho \in Q^{|w|+1}$. It has to satisfy that $\rho_1 \in I$ and for all $1 \leq i \leq |w|$ that

$\rho_{i+1} \in \delta(\rho_i, w_i)$. We call a run accepting if $\rho_{|w|+1} \in F$. If a word w generates an accepting run, we say the automaton accepts or recognizes w. We write $w \models \mathcal{M}$ and

$$L(\mathcal{M}) = \{w \in \Sigma^* \mid w \models \mathcal{M}\}.$$

In the deterministic case there is always exactly one run on each input.

Finite automata recognize the regular languages, for which we already showed characterizations in terms of finite syntactic monoids and MSO logic. Note that finite automata can be determinized. Also, deterministic automata can be minimized and then be used to compute the syntactic monoid: It is isomorphic to the transition monoid of the minimal automaton.

6.1.2 Pushdown Automata

In the Chomsky hierarchy, above the regular languages are the context-free languages (CFL). CFLs are the languages generated by context-free grammars and they coincide with those that are accepted by pushdown automata. A pushdown automaton (PDA) is a finite automaton enhanced by a pushdown storage. Usually, PDAs are defined to be accepting through emptiness of the stack. However, the model that accepts through final states is equivalent.

Definition 38 (Pushdown automaton)**.** *A non-deterministic pushdown automaton $\mathcal{M}$ is a tuple $(Q, I, \Sigma, \Gamma, \perp, \delta)$ where:*

- *Q is the finite set of states.*
- *$I \subseteq Q$ is the set of initial states.*
- *Σ is the input alphabet.*
- *Γ is the pushdown alphabet.*
- *$\perp \in \Gamma$ is the bottom-of-stack symbol.*
- *$\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times \Gamma \times Q \times \Gamma^*$ is the finite transition relation.*

If we want to define deterministic PDAs (DPDA), we have to resort to acceptance through final states because otherwise we do not get a meaningful model. We call a PDA or DPDA realtime if it does not perform ϵ-moves.

A configuration of a PDA $\mathcal{M}$ is a tuple $c \in Q \times \Sigma^* \times \Gamma^*$. The first component is the state the machine is in, the second is the word that is left to read and the third is the stack content. For two configurations $c = (q, w, \gamma)$ and $c' = (q', w', \gamma')$ we write $c \rightarrow_{\mathcal{M}} c'$ if there exists $(q, a, \gamma_1, q', g) \in \delta$ such that $w = aw'$ and $\gamma' = g\gamma_2 \ldots \gamma_{|\gamma|}$. A sequence of configurations that satisfies $\to_{\mathcal{M}}$ is called a run. Now, the language accepted by $\mathcal{M}$ if there exists an accepting run, i.e.

$$L(\mathcal{M}) = \{w \in \Sigma^* \mid (q, w, \bot) \to^*_{\mathcal{M}} (q', \epsilon, \epsilon),\ q \in I,\ q' \in Q\}.$$

Note that we allowed ϵ-moves here.

The languages accepted by PDAs are the context-free languages. CFLs are not closed under complementation and intersection, however, they are closed under union, homomorphisms and inverse homomorphisms. Emptiness is decidable and equivalence and universality are undecidable. Also, deterministic PDAs are strictly weaker than non-deterministic PDAs. Given a CFL L, we call L a linear language if there exists a PDA that, on each input of L, works in a way that no symbol ever gets pushed onto the stack after the first time an element has been popped off.

There is a relationship between regular tree languages and CFLs: The yield of a regular tree language is context-free and every CFL can be represented as a yield of a regular tree language. This relationship should, however, not suggest that the CFLs are the best fit for a counterpart of regular tree languages in the word domain. While building the yield we actually throw away the tree structure. This shows again in solving the word problem for CFLs where we have to reconstruct a parse tree. In contrast, when solving the word problem for regular tree languages, we have the full tree as input. Hence, the word problem for regular forest languages might have a lower complexity.

Counter automata are special PDAs and form a subset of the CFLs. We only consider the realtime version.

Definition 39 (Counter automaton). *A non-deterministic one-counter automaton (NOCA) $\mathcal{M}$ with threshold k is a tuple $(Q, I, F, \Sigma, \delta_1, \ldots, \delta_k)$ where:*

- *Q is the finite set of states.*
- *$I \subseteq Q$ is the set of initial states.*
- *$F \subseteq Q$ is the set of final states.*
- *Σ is the input alphabet.*
- *For all $h \in [k]$, $\delta_h \subseteq Q \times \Sigma \times Q \times \mathbb{Z}$ is the finite transition relation for height h.*

There also exist deterministic one-counter automata (DOCA).

A configuration of such an automaton consists of the state and the value of the counter. The counter, as we define it now, holds values of $\mathbb{N}$, i.e. does not become negative. Negative numbers, however, can be modeled easily by storing the sign in the states. A counter automaton $\mathcal{M}$ and a word $w \in \Sigma^*$ now induce a set of runs. A run ρ is an element of $(Q \times \mathbb{N})^*$. It has to satisfy that $\rho_1 = (q, 0)$ where $q \in I$ and that for $\rho_i = (q_1, m)$ and $\rho_{i+1} = (q_2, n)$ holds that $(q_2, n - m) \in \delta_{\min(k,m)}(q_1, w_i)$. A run is accepting if its last position contains a final state. A word is accepted if it induces an accepting run where we again write $w \models \mathcal{M}$ and $L(\mathcal{M}) = \{w \in \Sigma^* \mid w \models \mathcal{M}\}$.

For our purposes it is sufficient to only consider counter automata that in each step alter the counter value at most by 1. Examples for counter languages are the following: $\{a^n b^n \mid n \in \mathbb{N}\}^*$, $\{a^n b a^n c \mid n \in \mathbb{N}\}^*$, or $\{a^n b a^n \mid n \in \mathbb{N}\}^*$.

6.1.3 Turing Machines and Complexity

Turing machines are the most general devices to accept decidable languages. They have a finite state control and a tape as a storage. By restricting time or space consumption of machines measured in the input length, we get complexity classes. Important ones are **PSPACE**, **NP**, **P**, **NL**, and **L**; we assume the reader to be acquainted with basic complexity theory. Later we will also be interested in lower complexity classes, however, we will use circuits to capture those classes.

6.2 Finite Tree Automata

Finite tree automata generalize finite word automata and function in a very analogue way. An input tree generates a run that is now a tree that is structurally equivalent to the input tree. There are, however, two versions of tree automata, namely bottom-up (BUTA) and top-down (TDTA) tree automata. In TDTAs, the root is assigned some initial state and leaves correspond to final states whereas in BUTAs the order is reversed. We further distinguish between determinism and non-determinism as well as the ranked and unranked case. For BUTAs, determinism and non-determinism have equivalent power and for unranked trees only the BUTA model makes sense. Because of these facts, and the fact that deterministic TDTAs are weaker, we mainly focus on BUTAs.

Definition 40 (Bottom-up tree automaton for ranked trees)**.** *A non-deterministic bottom-up tree automaton $\mathcal{M}$ of rank $r\colon \Sigma \to \mathbb{N}$ is a tuple $(Q, I, F, \Sigma, (\delta_a)_{a\in\Sigma})$, where:*

- *Q is the finite set of states.*
- *$F \subseteq Q$ is the set of final states.*
- *Σ is the input alphabet.*

- $\delta_a \subseteq Q^{r(a)} \times Q$ *is the transition relation for* $a \in \Sigma$.

Given a tree t over the ranked alphabet (Σ, r) we define a run ρ as a tree over the ranked alphabet $(\Sigma \times Q, r \circ \pi_1)$, which is structurally equivalent to t, but the labels are complemented by a component that holds the state. Meanwhile, the rank is only be sensible to the first component. Consider an inner node v in ρ with children v_1 to $v_{r(a)}$ where a is the letter of v. If q is the state of v and $q_1, \dots, q_{r(a)}$ are the states of $v_1, \dots, v_{r(a)}$, then it must hold that $(q_1, \dots q_{r(a)}, q) \in \delta_a$. Note that in the case of leaves we assign the state $\delta_a() \in Q$ where a is the letter of a leaf, so $r(a) = 0$. This is the reason why there are no explicit initial states. If this condition holds in every state, then ρ is indeed a run. A run is *accepting* if the state of the root is in F.

Evaluating terms over finite single-sorted algebras is equivalent to computations of deterministic BUTA for ranked trees. Given a signature $\sigma \in \mathbb{N}^*$, let $\Sigma = \{\circledast_1, \dots, \circledast_{|\sigma|}\}$ and $r(\circledast_i) = \sigma(i)$. Consider a term $t \in \mathbb{T}(\sigma)$ now that we want to evaluate over an algebra $\mathcal{A} = (\mathbb{D}; \circledast_1, \dots, \circledast_{|\sigma|})$, note that t is also a tree of rank r. We define a deterministic BUTA having $\mathbb{D}$ as the state set and $\delta_{\circledast_i} = \circledast_i$ and obviously, the state the automaton computes in the end for the root is the evaluation of the term t. Taking it one step further, if $\mathcal{A}$ is recognizing some language, there is an accepting subset of $\mathbb{D}$. If we choose this accepting set as the final set of the automaton, we can see that the languages accepted by BUTAs over ranked trees coincide with languages recognized by finite single-sorted algebras.

In the unranked case, nodes in trees may have an arbitrary number of children. In this case we only consider the deterministic version. Actually we get an automaton model that accepts forests, however, we still call it a tree automaton.

Definition 41 (Deterministic bottom-up tree automaton for unranked trees)**.** *An unranked bottom-up tree automaton* $\mathcal{M}$ *is a tuple* $(Q, q_I, F, \Sigma, \delta_V, \delta_H)$*, where:*

- Q *is the finite set of states.*
- $q_I \in Q$ *is the initial state.*
- $F \subseteq Q$ *is the set of final states.*
- Σ *is the input alphabet.*
- $\delta_V \colon Q \times \Sigma \to Q$ *is the vertical transition relation.*
- $\delta_H \colon Q \times Q \to Q$ *is the horizontal transition relation. In particular it is the associative operation of a monoid* $(Q; \delta_H, q_I)$ *whose neutral element is* q_I.

A BUTA has two transition functions. One that collects the states of all children of some parent into a single state. For this to be well-defined we need associativity.

We also need a neutral element for the leaves. This makes the horizontal transition relation a monoid. Further, the vertical transition relation takes the label of the recent node and the collected state and then outputs a state for the recent node.

Formally, the semantic is again defined by a run ρ of $\mathcal{M}$ on the input tree t, which is structurally equivalent to t but this time over Q as the alphabet. A leaf in ρ is labeled by $\delta_V(q_I, a)$ where a is the letter of the corresponding leaf in t. A general node v in ρ with children v_1 to v_n labeled q_1 to q_n is labeled by $\delta(q, a)$ where q is the product $\delta_H(q_1, \dots, q_n)$ in the monoid $(Q; \delta_H, q_I)$. A run is accepting if the root of ρ has a label $q \in F$. In case of an input forest, the automaton accepts if the product of all root labels of the trees is in F.

BUTAs accept precisely the languages recognized by finite forest algebras [BW08].

6.3 Finite Nested Word Automata

Recall that nested words are words augmented with an additional matching information. The appropriate automaton model for these words should make use of the nesting without the need of a storage. This is achieved by having the automaton not reading the word from left to right but by reading the word guided by the matching. In finite automata for ordinary words, each step a letter is read and the state computed in the previous step is used to obtain the new state. In the case of nested word automata, we may not only access the state but also states computed after the matching position.

Definition 42 (Finite nested word automaton). *A non-deterministic finite nested word automaton $\mathcal{M}$ is a tuple $(Q, I, F, \Sigma, \delta, \delta_{\leadsto})$, where:*

- *Q is the finite set of states.*
- *$I \subseteq Q$ is the set of initial states.*
- *$F \subseteq Q$ is the set of final states.*
- *Σ is the input alphabet.*
- *$\delta \subseteq Q \times \Sigma \times Q$ is the transition relation.*
- *$\delta_{\leadsto} \subseteq Q \times \Sigma \times Q \times Q$ is the transition relation of positions being the right part of a matching.*

Such an automaton works just like a classical word automaton with the exception of positions i for which there exists a position j such that $j \leadsto i$. In this situation,

$\delta_\rightsquigarrow$ is used. It makes use of the state computed in position $j+1$. A deterministic version has transition functions $\delta\colon Q \times \Sigma \to Q$ and $\delta_\rightsquigarrow\colon Q \times \Sigma \times Q \to Q$.

Formally, given a nested word w with a matching predicate $\rightsquigarrow$ as input, the automaton $\mathcal{M}$ induces a run ρ, which is an element of $Q^{|w|+1}$. A run satisfies $\rho_1 \in I$. Also, if for $i \in [|w|]$ there does no j exist for which $j \rightsquigarrow i$, we have that $\rho_{i+1} \in \delta(\rho_i, w_i)$. Else, if such a j exists, we have that $\rho_{i+1} \in \delta(\rho_i, \rho_{j+1}, w_i)$. We use $j+1$ instead of j because we want to use the state that already has seen the letter in position j. A run is accepting if the last position contains an accepting state. Then we write $(w, \rightsquigarrow) \models \mathcal{M}$ and $L(\mathcal{M})$ is the set of all nested words accepted by $\mathcal{M}$.

Weakly nested words allow for positions that are part of a matching where the matching position is not part of the word. A finite nested word automaton can handle these by simply ignoring $\rightsquigarrow \infty$ and $-\infty \rightsquigarrow$.

Proposition 43. *Finite nested word automata accept precisely the regular nested word languages.*

We delay showing this proposition until the next section because we will pair it with the analogue statement for visibly pushdown automata.

6.4 Visibly Pushdown Automata

Nested words and well-matched words are equivalent models, differing in the fact that nested words have additional explicit matching information. Well-matched words are just ordinary words with special semantics. Therefore, the matching information is present in a more indirect way. That is why we do not have a finite automaton model for well-matched words, in contrast to nested words. Here, we need pushdown automata. In fact, a restricted version of pushdown automata sufficient to capture all regular sets of well-matched words. If we strictly define it as a PDA, we have the requirement that the kind of input letter already determines how the stack is accessed:

- For a call letter, one symbol is pushed.
- For a return letter one symbol is popped.
- For an internal letter, the stack is not accessed at all.

However, to get a more convenient model we give a new definition that does not use the definition of PDAs. Note that there is no bottom-of-stack symbol since we do not want the automaton to be able to read a return letter after a well-matched word is read. The automaton still has a way to remember when the stack is empty: By pushing an annotated symbol first and then maintaining this information.

Definition 44 (Visibly pushdown automaton)**.** *A non-deterministic visibly pushdown automaton (VPA) $\mathcal{M}$ is a tuple $(Q, I, F, \hat{\Sigma}, \Gamma, \delta_{\text{call}}, \delta_{\text{ret}}, \delta_{\text{int}})$, where:*

- *Q is the finite set of states.*
- *$I \subseteq Q$ is the set of initial states.*
- *$F \subseteq Q$ is the set of final states.*
- *$\hat{\Sigma} = (\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \Sigma_{\text{int}})$ is the visible input alphabet.*
- *Γ is the pushdown alphabet.*
- *$\delta_{\text{call}} \subseteq Q \times \Sigma_{\text{call}} \times Q \times \Gamma$ is the transition relation for call letters.*
- *$\delta_{\text{ret}} \subseteq Q \times \Sigma_{\text{ret}} \times \Gamma \times Q$ is the transition relation for return letters.*
- *$\delta_{\text{int}} \subseteq Q \times \Sigma_{\text{int}} \times Q$ is the transition relation for internal letters.*

Note that the nondeterministic transition relations can be also read, for example, as $\delta_{\text{call}} \colon Q \times \Sigma_{\text{call}} \to 2^{Q \times \Gamma}$. In the case of deterministic automata, the image is a set, which always has size one. Since such a relation is a function, we may write it equivalently as $\delta_{\text{call}} \colon Q \times \Sigma_{\text{call}} \to Q \times \Gamma$.

Such an automaton is called visibly because the input letter always indicates what happens on the stack. Languages accepted by VPAs are the visibly pushdown languages VPL.

Given a well-matched input word $w \in \text{WM}(\hat{\Sigma})$ and a VPA $\mathcal{M}$, a run ρ is a word in $(Q \times \Gamma^*)^{|w|+1}$ where $\rho_1 = (q, \epsilon)$ and $q \in I$. Further, if $w_i \in \Sigma_{\text{call}}$, then $\rho_i = (q, \gamma)$ and $\rho_{i+1} = (q', a\gamma)$ where $(q', a) \in \delta_{\text{call}}(q, w_i)$. If $w_i \in \Sigma_{\text{ret}}$, then $\rho_i = (q, a\gamma)$ and $\rho_{i+1} = (q', \gamma)$ where $q' \in \delta_{\text{ret}}(q, w_i, a)$. Finally, if $w_i \in \Sigma_{\text{int}}$, then $\rho_i = (q, \gamma)$ and $\rho_{i+1} = (q', \gamma)$ where $q' \in \delta_{\text{int}}(q, w_i)$. A run is accepting if the last position is of the form (q, ϵ) for $q \in F$; we write $w \models \mathcal{M}$ and $L(\mathcal{M}) \subseteq \text{WM}(\hat{\Sigma})$ is the set of all accepted well-matched words.

VPAs can be determinized [AM04]. The result of the determinization procedure can serve as a useful normal form. A sketch of this construction is as follows. First, note that the powerset construction alone does not work. As a state set for the deterministic machine we choose $Q' = 2^{Q \times Q}$ for Q being the state set of the original automaton. The main idea is the following: When reading an input word w, let $q \in Q'$ be a state that is reached after reading the first k letters. Now, q holds the information for each pair of states $(q_1, q_2) \in Q \times Q$ whether $q_1 \xrightarrow{u} q_2$ exists where

u is the maximal well-matched suffix of $w_1 \dots w_k$. To maintain this information we choose the stack alphabet to be $\Gamma' = Q' \times \Sigma_{\text{call}}$. When reading a call letter we store it onto the stack together with the recent state. After reading a call letter, the maximal well-matched word that comes before is ϵ, because well-matched words never end in call letters. So, we jump into a state that represents the identity map. When a return letter is read, we have all the information present to maintain the semantic of the state. Let the well-matched word be $uavb$ for u and v being well-matched, $a \in \Sigma_{\text{call}}$, and $b \in \Sigma_{\text{ret}}$. Now, if the automaton is about to read b it is in a state that holds the information for v. Through the stack it has access to the state that holds the information for u. The letter a is also present. So, the state for $uavb$ can be computed.

Proposition 45. *VPAs recognize precisely the visibly pushdown languages.*

Proof. Given a language L accepted by a determinized VPA, we define a finite extend algebra that recognizes L. First, consider $\delta_{\text{ret}} \subseteq Q \times \Sigma_{\text{ret}} \times \Gamma \times Q$. Following the determinization construction we can actually see δ_{ret} as being of the form $\delta_{\text{ret}} \subseteq Q \times \Sigma_{\text{call}} \times Q \times \Sigma_{\text{ret}} \to Q$. Recall that the states of Q are sets of pairs of states of the original automaton. So, for δ_{ret} we have $(q, a, q', b) \mapsto q \circ f_{a,b}(q')$. We see that Q together with $\circ$ forms a monoid. This is the base for the finite extend algebra. So, we choose Q to be the domain and $\circ$ to be the binary operation, i.e. Q is the horizontal monoid. Further, for all $(a, b) \in \Sigma_{\text{call}} \times \Sigma_{\text{ret}}$ the function $\triangle_{a,b}$ is a unary operation in the algebra. Internal letters can be treated in the obvious way resulting in another set of 0-ary operations $\diamond_c$. Correctness of the construction can be seen by induction over the structure of well-matched words.

For the reverse direction, suppose a language of well-matched words L for which the syntactic extend algebra $\text{Synt}(L) = (H; +, (\triangle_{a,b})_{a \in \Sigma_{\text{call}}, b \in \Sigma_{\text{ret}}}, 0, (\diamond_c)_{c \in \Sigma_{\text{int}}})$ is finite. The automaton we construct has H as a state set. We define δ_{ret} to be $(q, a, q', b) \mapsto q + \triangle_{a,b}(q')$. Also, we define δ_{int} as $(q, c) \mapsto q + \diamond_c$. Finally, δ_{call} stays as described in the determinization procedure. Again, correctness follows by induction. □

Proposition 43 follows from the previous proof since nested word automata and VPAs are equivalent. Also, the determinization procedure can be applied with minimal alternation to nested word automata, leading to a very similar direct proof. This leads a normal form for nested word automata.

Matched words, in contrast to well-matched ones, relax the condition that every call letter needs a matching return letter. We defined VPAs to be only accepting well-matched words. Originally, VPAs were defined for this more relaxed version. However, then we would not get the correspondence to algebra as we did before. Besides that, everything can be generalized to the matched case.

At this point we survey basic properties of the class of VPLs [AM04]. First, when $\hat{\Sigma}$ is fixed, VPLs are closed under intersection, union, concatenation, and Kleene star. They are also closed under complementation against $\mathrm{WM}(\hat{\Sigma})$ but not against Σ^*. Closure under homomorphisms is given only for those that are compatible with visibility. For example, let L be a VPL over $\hat{\Sigma} = (\{a\}, \{b\}, \emptyset)$ and ϕ be a homomorphism ϕ. If it holds that $\Delta(\phi(a)) \geq 0$ and $\Delta(\phi(a)) = -\Delta(\phi(b))$ where the height profiles of $\phi(a)$ and $\phi(a)$ do not go below 0, then $\phi(L)$ is also a VPL.

The situation for decidability is also pleasant: Given VPAs $\mathcal{M}_1$ and $\mathcal{M}_2$, it is decidable whether $L(\mathcal{M}_1) = L(\mathcal{M}_2)$, $L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$, as well as $L(\mathcal{M}_1) = \emptyset$. The decidability results and closure properties demonstrate that VPLs might offer a better trade-off between those properties and expressibility than CFLs. They behave much tamer, similar to the regular word languages, which comes as no surprise. So, they are a much more meaningful counterpart than the CFLs.

6.5 Within Visibly Pushdown Languages

Visibly pushdown languages have sparked great interest as they bring the world of trees and words together. They provide a good tradeoff between expressibility on the one hand and closure and decidabilities on the other hand. There are also close relations to terms and term evaluation. Soon after their discovery a line of research emerged that tried to generalize results known for regular word languages to visibly pushdown languages. Often, this is successful, but sometimes it is not. In those cases it is beneficial to approximate results by showing them for subclasses of VPLs. We will consider some subclasses and also look at an interesting open problem.

6.5.1 Very Visibly Pushdown Languages

The first subclass of the visibly pushdown languages are the *very visibly pushdown languages*[1]. A VPA decides depending on the input letter whether a symbol is pushed onto or popped off the stack. In the very visibly case, if a call letter is read, also the symbol that is pushed is already determined by the letter read. We generalize this model slightly by allowing the automaton to know the stack height up to some threshold k, similarly as it is the case for counter automata.

Definition 46 (Very visibly pushdown automaton with threshold k (k-VVPA))**.** *A non-deterministic very visibly pushdown automaton $\mathcal{M}$ with threshold k is a tuple $(Q, I, F, \hat{\Sigma}, \Gamma, \delta_{\mathrm{call}}, \delta_{\mathrm{ret}}, \delta_{\mathrm{int}})$, where:*

[1]This naming might seem awkward, but it was chosen to be consistent with the pre-existing name of *visibly pushdown language*

- Q *is the finite set of states.*
- $I \subseteq Q$ *is the set of initial states.*
- $F \subseteq Q$ *is the set of final states.*
- $\hat{\Sigma} = (\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \Sigma_{\text{int}})$ *is the visible input alphabet.*
- Γ *is the pushdown alphabet.*
- $\delta^i_{\text{call}} \subseteq Q \times \Sigma_{\text{call}} \times Q \times \Gamma$ *is the transition relation for call letters and* $i \in \{0, \ldots, k\}$*. We also require that for all* $q_1, q_1', q_2, q_2' \in Q$*,* $a_1, a_2 \in \Sigma_{\text{call}}$*, and* $\gamma_1, \gamma_2 \in \Gamma$ *with* $(q_1, a_1, q_1', \gamma_1) \in \delta^i_{\text{call}}$ *and* $(q_2, a_2, q_2', \gamma_2) \in \delta^i_{\text{call}}$ *holds that* $a_1 = a_2 \Rightarrow \gamma_1 = \gamma_2$.
- $\delta^i_{\text{ret}} \subseteq Q \times \Sigma_{\text{ret}} \times \Gamma \times Q$ *is the transition relation for return letters and* $i \in \{0, \ldots, k\}$.
- $\delta^i_{\text{int}} \subseteq Q \times \Sigma_{\text{int}} \times Q$ *is the transition relation for internal letters and* $i \in \{0, \ldots, k\}$.

The semantics here are very similar to the one for VPAs. The only difference is that if in a configuration the stack has height α, the transition relations $\delta^{\max(\alpha,k)}_{\text{call}}$, $\delta^{\max(\alpha,k)}_{\text{ret}}$, and $\delta^{\max(\alpha,k)}_{\text{int}}$ are used. A k-VVPA is a VVPA with threshold k and by a k-VVPL denote a language accepted by some k-VVPA. By VVPL we address a k-VVPL for some $k \in \mathbb{N}$.

It is easy to see that every language accepted by some k-VVPA is also accepted by a VPA. In order to see this, one can build a VPA $\mathcal{M}'$ based on some k-VVPA $\mathcal{M}$ that maintains the stack height up to k in the stack content; if $\mathcal{M}$ has Γ as stack alphabet, $\mathcal{M}'$ will have $\Gamma \times \{0, \ldots, k\}$ as stack alphabet.

In k-VVPAs it is possible to assume that $\Gamma = \Sigma_{\text{call}}$, where, every time some $a \in \Sigma_{\text{call}}$ is read, a is pushed onto the stack. That way the maximal information is stored.

An example of a language that is a VPL but not a k-VVPL is

$$\{a^m b^n a^o b^o c b^{m-n} \mid m, n, o \in \mathbb{N},\ m \geq n\}$$

for $\Sigma_{\text{call}} = \{a\}$, $\Sigma_{\text{ret}} = \{b\}$, and $\Sigma_{\text{int}} = \{c\}$. The idea is that an automaton reading the word has to remember when the prefix $a^m b^n$ is read, since that is the height where on the matching side the letter c has to be present. A k-VVPA cannot store this information onto the stack and is, therefore, left with its states. By using a pumping argument, one can see that this information cannot be maintained through reading the factor $a^o b^o$.

We can see how the threshold works by modifying the previous example. The language

$$L_i = \{a^m b^n a^o b^o c b^{m-n} \mid m, n, o \in \mathbb{N},\ 0 \leq m - n < i\}$$

is accepted by an i-VVPA but not by an $(i-1)$-VVPA.

Another example for a language that can be recognized by VVPAs is the Dyck language $\mathcal{D}_p \subseteq \{a_1, \ldots, a_p, b_1, \ldots, b_p\}^*$ where the letters a_i are call and b_i are return letters for $i \in [p]$. For $i = 1$, the set $\mathcal{D}_i$ is just the set of well-matched words over two letters. In general, $\mathcal{D}_i$ is the set of well-matched words for which it holds that each position with letter a_i matches a position with letter b_i. So, $\mathcal{D}_p$ can be regarded as the set of all well parenthesized expressions using p pairs of parentheses. The Dyck language is recognized by a VVPA for any number of parentheses.

A basic property of VVPAs is captured by the following proposition:

Proposition 47. *Very visibly pushdown automata can be determinized.*

Proof. Let $\tau_k \colon \hat{\Sigma} \to (\Sigma_{\text{call}} \cup \Sigma_{\text{ret}} \times \Sigma_{\text{call}} \cup \Sigma_{\text{int}}) \times [k]$ be a transduction that takes a well-matched word and labels each letter its height up to k and each return letter its matching call letter. An example for this would be $\tau_1(a_1 a_2 a_1 bbb) = (a_1, 0)(a_2, 1)(a_1, 1)(b, a_1, 1)(b, a_2, 1)(b, a_1, 0)$ where $a_1, a_2 \in \Sigma_{\text{call}}$ and $b \in \Sigma_{\text{ret}}$. For each VVPA $\mathcal{M} = (Q, I, F, \hat{\Sigma}, \Gamma, \delta_{\text{call}}, \delta_{\text{ret}}, \delta_{\text{int}})$ there exists an NFA $M = (Q, I, F, (\Sigma_{\text{call}} \cup \Sigma_{\text{ret}} \times \Sigma_{\text{call}} \cup \Sigma_{\text{int}}) \times [k], \delta)$ with:

- For $a \in \Sigma_{\text{call}}$ it holds $(q, (a, i), q') \in \delta$ if there exists γ such that $(q, a, q', \gamma) \in \delta^i_{\text{call}}$.
- For $b \in \Sigma_{\text{ret}}$ and $a \in \Sigma_{\text{call}}$ it holds $(q, (b, a, i), q') \in \delta$ if there exists γ such that $(q, b, \gamma, q') \in \delta^i_{\text{ret}}$ and γ is the letter that is pushed if a is read.
- For $c \in \Sigma_{\text{int}}$ it holds that $(q, (c, i), q') \in \delta$ if $(q, c, q') \in \delta^i_{\text{int}}$.

Now $\mathcal{M}$ accepts a word w if and only if M accepts $\tau(w)$. The automaton M can be determinized. Subsequently, we can do the reverse construction and obtain a VVPA from the DFA, which then in turn is deterministic. In other words: VVPAs can be determinized by the powerset construction. □

Finally, consider the example language $L \subseteq \{a_1, a_2, b, c_1, c_2\}$ where $\Sigma_{\text{call}} = \{a_1, a_2\}$, $\Sigma_{\text{ret}} = \{b\}$, and $\Sigma_{\text{int}} = \{c_1, c_2\}$, which is defined through the following grammar rules $S \to a_1 S b c_1 \mid a_2 S b c_2 \mid \epsilon$. This language clearly is recognized by a VVPA. Whenever it reads some b, it receives the information from the stack whether the next symbol has to be c_1 or c_2. Now, look at the reversal L^R of L where call letters become return letters and vice versa. There is no VVPA for L^R since it has no way of storing whether it read a c_1 or c_2 on some stack height because these letters are internal.

So, on the matching side the information to decide between b_1 and b_2 is not present, and therefore VVPLs are not closed under reversal.

6.5.2 Visibly Counter Languages

By restricting the automaton model even further, we move from VVPAs to visibly counter automata (VCA). They also have a height test up to some threshold, which is similar to NOCAs and DOCAs. This model appears in [BLS06, KLL15b, HKLL15, KLL15a].

Definition 48 (Visibly Counter Automaton)**.** *A k-VCA is a k-VVPA for which the stack alphabet Γ has cardinality one.*

In the case of k-VCAs, the definition of the transition relations can be collapsed to $\delta^i_{\text{call}} \subseteq Q \times \Sigma_{\text{call}} \times Q$, $\delta^i_{\text{ret}} \subseteq Q \times \Sigma_{\text{ret}} \times Q$, and $\delta^i_{\text{int}} \subseteq Q \times \Sigma_{\text{int}} \times Q$, and so it can be even further collapsed to $\delta^i \colon Q \times \Sigma \times Q$.

An equivalent definition would be to impose a visibility restriction onto NOCAs. The set of languages accepted by k-VCAs is k-VCL and VCL is the union over all $k \in \mathbb{N}$.

The proof of Proposition 47 also works for VCAs:

Proposition 49. *Visibly counter automata can be determinized.*

A prime example for a language that is a VCL is $\mathcal{D}_1$. Further, $\mathcal{D}_i$ is not a VCL for $i > 1$. For some k the language $\{a^n b^{n-k} a^{m-k} b^m \mid n, m > k\}$ has a $k+1$-VCA but no k-VCA.

Proposition 50. *The visibly counter languages are closed under reversal.*

Proof. When performing the construction of the proof of Proposition 47. One can modify the resulting automaton M such that it accepts the reverse. The result can be translated back into a VCA. □

6.5.3 Intersection Problems

As already indicated, VPLs can be used to express term evaluation over finite algebras. This motivates some questions, amongst others: If X is some low complexity class, what is $\mathsf{VPL} \cap X$. In the last chapter we go more into the details of this aspect. However, for now, we investigate a closely related question, which in its most general form asks for decidability of the following: Given two VPLs L_1 and L_2, does a regular language R exist, such that $L_1 = L_2 \cap R$. This problem has been proven to be undecidable [Kop16].

However, there are interesting special cases, where we get decidability or we do not know about decidability. These special cases may have practical applications.

To that end we refine our notion of matching. In general, a call letter matches a return letter. A restriction is present in the Dyck languages in which each open parenthesis has exactly one matching closing one. We can take this even further and define a bipartite graph $G = (\Sigma_{\text{call}} \cup \Sigma_{\text{ret}}; E)$ over the partition sets Σ_{call} and Σ_{ret}. If there is an edge $(a, b) \in E$, it means that we allow that a position with an a may match a position with a letter b. In this case we call G a *matching graph*. Given a matching graph G, we call a word w *strongly well-matched* with respect to G if all its matchings respect G. The set of all strongly well-matched words with respect to G is denoted as $\text{SWM}(G)$. One can observe now, for example, that $\mathcal{D}_k$ equals $\text{SWM}(G)$ for $G = (\Sigma_{\text{call}} \cup \Sigma_{\text{ret}}; E)$, where $(a_i, b_j) \in E$ if and only if $i = j$. The set of all well-matched words we get by taking the maximal matching graph and denote this set by WM.

We can ask the following question: Given some VPL L, is it a regular restriction of the set of strongly well-matched words? In other words: Answering this question may have applications, for example, in parsing XML. An XML document is a strongly well-matched word in which many opening parentheses have one closing one if one takes parameters into account. If one abstracts those away, an XML document becomes a subset of a Dyck language. Now, suppose we know that we have a language at hand, that is the intersection of a regular language and a strongly well-matched set. In this case we can split the parsing. Under the premise that the input word is valid, i.e. is strongly well-matched, checking validity is nothing different than solving a word problem for a regular language and this is a problem for which a rich set of tools already exists.

This problem is still open, but attempts have been made. In [BLS06] a partial solution can be found for the problem of deciding whether some VPL L is the regular restriction of the set of well matched words, i.e. does a regular language R exist, such that $L = R \cap \text{WM}$. The result is obtained in two steps. First, it is decided whether for L there exists some k such that L is recognized by some k-VCA. The second step is deciding whether k can be reduced. If k can be reduced to 0, the answer is yes: A language is accepted by some 0-VCA if and only if it is the intersection of a regular language and WM. In a 0-VCA, there is no hight test that impacts the states. So, the transitions the automatons perform are just those of a finite automaton. The counter only enforces that the input is well-matched.

Now, consider the problem of deciding whether a VPL L is of the form $L = R \cap \text{SWM}(G)$ for a regular language R and a matching graph G. It is not difficult to find G. First, there exists a unique minimal G with respect to the number of edges. It should contain exactly those matchings that occur in some word in L. Hence, we may just search for the smallest G and try to find a fitting R. There are only $2^{|\Sigma_{\text{call}}|\cdot|\Sigma_{\text{ret}}|}$ different matching graphs, which is finite. Since inclusion is decidable for VPLs, we can check $L \subseteq \text{SWM}(G)$ for all G and then choose the smallest one.

Finding R is harder and we do not have a solution for this problem yet. However, one could try top mimic the proof strategy of [BLS06]. Instead of finding a k-VCA and then a 0-VCA, one could try to find a k-VVPA and then a 0-VVPA. If we have a 0-VVPA, we still need to check whether L is actually a regular restriction of $\text{SWM}(G)$.

For a certain special case, checking whether a 0-VVPA accepts a language that is a regular restriction of $\text{SWM}(G)$ is actually already doable. If G is a matching graph in which every return letter has at most one matching call letter, then, if the language being a subset of $\text{SWM}(G)$ is accepted by some 0-VVPA, it is already a regular restriction. The idea is that storing content on the stack does not provide any additional information for the time the matching return letter is read, since the call letter is already determined by the return letter. So, we can distill a finite automaton out of the 0-VVPA the same way as in the case of 0-VCA by simply ignoring the stack.

Now, suppose we have a matching in which each call letter has at most one matching return letter. This case can be handled by first building an automaton for the reverse language. If the language is a regular restriction, then there exists a 0-VVPA for the reverse language. After that step we can distill the finite automaton and again do a reversal to obtain the original language again.

Next steps could be to look at more complicated matchings until we obtain a method that works for all matching graphs. Of course, the step of converting VPAs to 0-VVPAs in the first place is also still open. This problem could be tackled using determinized VPAs. These are already very close to VVPAs in that only in certain situations they store more information than the call letter they just read to the stack.

The problem of deciding regular restrictions is related to questions we address in the final chapter where we analyze which VPLs are in certain low complexity classes. Knowing about decidability of the problem might help to solve problems over there. In particular a variant would be interesting where we do not ask for a regular language R such that $L = R \cap \text{SWM}(G)$ but for an aperiodic or quasiaperiodic regular language R.

6.6 Conclusion

Summary

After covering algebra and logic for capturing languages we considered automata in this chapter. We looked at versions for finite words, ranked trees, and forests first and presented the typical finite automaton models. Later, nested words and their finite automaton model were covered. Nested words are in correspondence to well-matched words, but the corresponding automaton model needs storage, which leads to visibly pushdown automata - a well behaving special kind of pushdown automaton.

VPAs being well-behaved means that many desirable properties of finite word automata are inherited. We are interested in finding more of these good properties. This is not always easy, so we defined intermediate classes between the VPLs and the regular languages. First, very visibly pushdown languages are recognized by very visibly pushdown automata, which are VPA that have the property that the call letter read determines the letter pushed onto the stack. A further restriction is to make the pushdown storage a counter. We then arrive at the visibly counter languages.

For the different automaton models we showed regularity in the sense that the languages recognized coincide with those recognized by finite algebras. One approach that we did not pursue is to capture VVPLs and VCLs by typed monoids.

In the end we looked at intersection problems and pointed out some first steps to solve them.

Contributions

In this chapter we chiefly surveyed existing concepts. However, we introduced one that is new: Very visibly pushdown automata are a natural intermediate model between VCAs and VPAs. This could be used to lift the proof strategy from [BLS06] to the case of strongly well-matched words. We laid out how such a proof could work.

Sources and Related Work

For basics on automata theory for we refer to [Str94, HU79, Tho97]. In addition, for tree automata see [CDG$^+$07].

VPAs and nested word automata appeared together as two sides of the same coin and were introduced by Alur and Madhusudan [AM04, AM09]. VPA had an incarnation before under the name of input-driven automata introduced by Mehlhorn [Meh80]. Most recent research, however, can be traced back to the paper of Alur and Madhusudan.

VCAs first appeared as a tool in [BLS06] for showing decidability of regularity questions. This is the kind of questions we addressed in the last section of this chapter. Recently it has been shown [Kop16] that the more general problem of deciding whether some VPL is the regular restriction of another VPL is not decidable.

VPAs are an attempt for a better tradeoff than pure PDAs. Litte expressibility is sacrificed while many good properties emerge. We want to mention another possibility, the so-called height deterministic automata [NS07, LMM09].

Further Research

The intersection problems we covered yield the most obvious thread for further research in this chapter. One could take up the proof strategy we outlined, which is rooted in [BLS06].

Chapter 7

Quantitative Automata

Up to this point we have mainly spoken about languages, which are either understood as a set of structures or as a subset of some free algebra.

A language $L \subseteq D$ can be interpreted as a map $D \to \{0, 1\}$ where D is the set of all structures or the domain of a free algebra. This map is the characteristic function of L. One major aspect of this work is to go beyond languages. That is, we not only consider problems that have a yes/no answer but problems where some richer output is desired. A prime example is to extend the characteristic function to $D \to \mathbb{N}$. The natural numbers as a target data type will appear in many places throughout this work. The following section as an example for this case.

7.1 Counting

We already discussed ordinary Boolean automata in the previous chapter. These automata can be called Boolean since they basically compute a single output bit. While maintaining the syntax of the different automata models, one can assign a generalized semantic, which in turn yields more information than just one bit indicating acceptance or rejection. We can assign a function $D \to \mathbb{N}$ to non-deterministic machines that assigns each input the number of accepting runs. Therefore, the image of the function is 0 if the input is rejected and positive if it is accepted. This shows that the Boolean case embeds into this setting, which we call *counting*. In principle we can apply this to any nondeterministic automaton. For deterministic and unambiguous automata the counting function and the characteristic function coincide.

When considering Turing machines and complexity, we obtain counting complexity classes by taking the set of counting functions corresponding to some non-deterministic complexity class. The set $\#\mathbf{P}$ contains all functions $f\colon \Sigma^* \to \mathbb{N}$ for which a non-deterministic poly-time Turing machine exists which, on input w, has $f(w)$ accepting computations. The same way we define $\#\mathbf{L}$.

For other classes, we borrow the #-notation and write, e.g., $\#\mathsf{NFA}$ or $\#\mathsf{VPA}$ for the sets of functions we get through counting accepting runs in NFAs or VPAs.

7.2 Weighted Automata

Counting in automata can be considered as extracting information about how the automaton works. If we want to implement functions that occur in application, we need to go beyond that and enhance our automaton model. Weighted automata are an active field of research that deals with non-deterministic automata that are equipped with a semiring $(R;+,\times)$. Each transition rule of the non-deterministic automaton is assigned an element of R. The automaton can then be assigned a function $D \to R$ in the following way: For each run, the weights that occur on its transitions are multiplied. That way we get one value for each run. These values are then being summed up. This sum is the output. Due to distributivity we can also view the problem from a different angle. Rather than the set of runs, consider the execution tree. The set of maximal paths in the execution tree yields the set of runs. Now, we evaluate this tree under the semiring as follows: A vertical step corresponds to multiplying the assigned value. We do this for all children of a node. Then all these values are being added, which is the horizontal operation.

It is easy to see how counting as defined previously embeds into the weighted framework: Take the semiring $(\mathbb{N};+,\times)$ and assign each transition the value 1. Then counting is achieved.

The idea of weighted automata, as we outlined above, is directly applicable to finite word automata. There exists a sizable body of work that considers other automata models under the weighted framework, like weighted tree automata and weighted visibly pushdown automata. We omit going into detail since we will cover an even more general framework in the following section.

7.3 Cost Functions and Cost Register Automata

A rather recent generalization of weighted automata are cost register automata (CRA). A CRA for finite words is an ordinary deterministic automaton equipped with a set of registers. The register automaton is a classical model, which is Turing complete. CRAs, however, are less general. In a CRA, the behavior of the states, i.e. the state transitions and register actions are totally data independent. That means that the same action is performed, no matter what values the registers hold. In particular, there is no 0-test. This restriction makes this model tractable while still having a significant expressibility - a situation that also occurs in the visibly pushdown model. Later we will bring these two models together.

More formally, a CRA is based on some algebra $\mathcal{A} = (\mathbb{D}; O)$ and, in the case of words, it implements a function $\Sigma^* \to \mathbb{D}$ or in general a function $D \to \mathbb{D}$. Note that in contrast to the weighted case, $\mathcal{A}$ does not have to be a semiring. In fact, $\mathcal{A}$ can be totally arbitrary. Further, a weighted automaton uses non-determinism to involve both operations of the semiring. CRAs on the other hand are deterministic. For every transition rule and register there exists a term that recombines old register values to obtain the new one.

No matter whether we consider words or other models, basically a CRA has a finite set of registers X and initial values for each register. When the input is read, each register value is updated according to state and letter read. In the end a final cost function is applied to recombine the register values to the single output value. For many considerations, it is beneficial to separate the actions of the transitions and the actual values. If we remove the initial value and the final cost function, we can regard a CRA as a device assigning each word of Σ^* a function $\mathbb{D}^X \to \mathbb{D}^X$. Here, a map $X \to \mathbb{D}$ is a valuation of the registers. This function again is composed of the functions each transition is assigned. Later, we will discuss this more thoroughly, but for now keep both views in mind.

The cost register models we define are always based on a single-sorted algebra. We do this solely because of succinctness. The cost register framework works just as well with many-sorted algebras. In this setting each register is assigned a sort.

In the following section we survey CRAs on words, which is the first kind of cost register machine introduced in the literature. We will use it as a blueprint for other kinds of cost register machines.

Also note that the complexity of CRAs is a major topic of interest, however, we delay complexity questions until Part II of this work.

7.3.1 Cost Register Automata for Finite Words

Initially the goal was to design an automaton models that models a function assigning each element of Σ^* a value. The value and the computations are based on some single-sorted algebra that is fixed for the automaton. The automaton has a set of registers that are updated according to the state and the letter read. The register updates the automaton performs can be captured by the following algebra:

Definition 51 (Register algebra)**.** *Given a single-sorted signature σ and a finite set of registers X, the register algebra is defined as $R_\sigma^X = \big((\mathbb{T}^X(\sigma))^X; \odot\big)$. The domain consists of functions $X \to \mathbb{T}^X(\sigma)$. The images of the functions consist of terms with variables of X. These terms can be interpreted as functions of the form $(\mathbb{T}^X(\sigma))^X \to \mathbb{T}^X(\sigma)$. The operation $\odot$ is then defined following this interpretation as $(a \odot b)(x) = (b(x))(a)$ for $a, b \in \mathbb{T}^X(\sigma))^X$ and $x \in X$.*

The operation $\odot$, which we defined in the algebra, models substitution of terms with variables. A term t of $\mathbb{T}^X(\sigma)$ represents also a function that takes a function $f\colon X \to \mathbb{T}^X(\sigma)$ and then each variable $x \in X$ in t by $f(x)$. The definition of $\odot$ models this substitution. For the product $(a \odot b)(x) = (b(x))(a)$ where $x \in X$, we first consider $b(x)$, where $b\colon X \to \mathbb{T}^X(\sigma)$ assigns each register a term. Then $b(x) \in \mathbb{T}^X(\sigma)$, but we can also interpret $b(x)$ as a map $(\mathbb{T}^X(\sigma))^X \to \mathbb{T}^X(\sigma)$ by the interpretation described above. As $b(x)$ is such a map, we may insert a into it, and thus have $(b(x))(a) \in \mathbb{T}^X(\sigma)$, hence $x \mapsto (b(x))(a)$ is a function $X \to T^X(\sigma)$.

Definition 52 (Cost register automaton for words (CRA))**.** *A cost register automaton $\mathcal{M}$ over a single-sorted algebra $\mathcal{A} = (\mathbb{D}; O)$ of signature σ is a tuple $(Q, q_I, \Sigma, \delta, X, \nu_0, \rho, \mu)$, where:*

- *Q is the finite set of states.*
- *$q_I \in Q$ is the initial state.*
- *Σ is the alphabet.*
- *$\delta\colon Q \times \Sigma \to Q$ is the transition function.*
- *X is the finite set of registers.*
- *$\nu_0\colon X \to \mathbb{D}$ is the initial register valuation.*
- *$\rho\colon Q \times \Sigma \to R_\sigma^X$ is the register update function.*
- *$\mu\colon Q \to \mathbb{T}^X(\sigma)$ is the final cost function.*

A CRA $\mathcal{M}$ implements a function

$$F_{\mathcal{A}}(\mathcal{M})\colon \Sigma^* \to \mathbb{D}.$$

Let $q_0 \dots q_{|w|} \in Q^*$ be the run of $\mathcal{M}$ on an input $w \in \Sigma^*$ where $q_0 = q_I$ is the initial state. We assign each step a valuation of the registers. The initial valuation is $\nu_0\colon X \to \mathbb{D}$. Now, we assume ν_{i-1} is already computed, then

$$\nu_i(x) = \mathrm{eval}_{\mathcal{A}}^{\nu_{i-1}}(\rho(q_{i-1}, w_i)(x)).$$

After w is read we arrive at a valuation $\nu_{|w|}$. The final output then is

$$F_{\mathcal{A}}(\mathcal{M})(w) = \mathrm{eval}_{\mathcal{A}}^{\nu_{|w|}}(\mu(q_{|w|})).$$

We defined the semantics in such a way that values of $\mathcal{A}$ are computed in each step, which involves evaluation in each step, but it is also possible to have another order where an evaluation over the target algebra only occurs once at the end. A CRA can be regarded as a device that generates a term. This term is evaluated over an algebra with given initial values. The term a CRA $\mathcal{M}$ generates is $F_{\mathcal{T}(\sigma(\mathcal{A}))}(\mathcal{M})$ if we choose the initial valuation as the identity function. Then we get

$$F_{\mathcal{A}}(\mathcal{M})(w) = \mathrm{eval}_{\mathcal{A}}^{\nu_0}(F_{\mathcal{T}(\sigma(\mathcal{A}))}(\mathcal{M})(w)).$$

The function $F_{\mathcal{T}(\sigma(\mathcal{A}))}(\mathcal{M})$ in turn can be obtained by the register algebra. The run $r \in Q^*$ induced by the input word $w \in \Sigma^*$ and the input itself lead to a sequence of register updates $\rho_i = \rho(r_i, w_i)$. Now, $(F_{\mathcal{T}(\sigma(\mathcal{A}))}(\mathcal{M})(w) = (\bigodot_{i \in [|w|]} \rho_i) \odot \mu(q_{\mathrm{final}})$ is the term the automaton computes.

Important examples for CRAs are those over $(\mathbb{N}; +)$ or $(\mathbb{N}; +, \times)$[1]. We may also use a free monoid as an algebra and by doing so get transducer-like automata.

One motivation for CRAs has been the need to obtain a model that generalizes weighted automata, which CRAs indeed do [ADD$^+$11]. There is also a tight relation to counting. For this statement we allow NFAs to have ϵ-transitions. This is useful for capturing the initial values, especially if the input is the empty word. Alternatively we could have restricted all initial values of the CRA to zero.

Note that automata using the algebra $(\mathbb{N}; +)$ may still contain multiplication in their register update terms, that is multiplication with a constant. The reason is that such multiplications can be replaced by a term of a fixed length only using addition.

Theorem 53. *The set of functions in* #NFA *coincides with functions implemented by CRAs over* $(\mathbb{N}; +)$*.*

Proof. This proof is related to a proof in [ADD$^+$11]. First, we show that each function in #NFA is in implemented by a CRA over $(\mathbb{N}; +)$. Given an NFA M, we construct a CRA N such that it implements the counting function of M. This CRA has only one state and one register for every state in M. Let δ be the transition function of M, then the register update function ρ of N is defined as follows: Let z be the single state of N, $a \in \Sigma$ and q be a register, which is also a state of M. Then $\rho(z, a, q)$ is the term that sums up all registers/states q' with $q \in \delta(q', a)$. The

[1]Note that in order to stay consistent with canonical notation, we omit mentioning the constant operations in the notation.

initial register valuation assigns all initial states the value 1 and all other states the value 0. The final cost function sums up all registers that are final states of M.

For the reverse we show that each function implemented by a CRA over $(\mathbb{N}; +)$ is in $\#\mathsf{NFA}$. Given a CRA N we construct the NFA M. Let Q be the state set of N and X the set of registers. Initially, M will have the state set $Q \cup Q \times X$. The number of paths reaching a state (q, x) in M will be the same as the register value of x.

In a state q we get to q' by reading the letter a and the register value for x is set to $v_1x_1 + v_2x_2 + \ldots + v_kx_k + c$, then in M for all $i \in [k]$ we insert v_i copies of the transition $((q, x_i), a, (q', x))$. We also insert c copies of the transition $(q, a, (q', x))$. Since δ is not a multiset, formally we have to implement the multiple copies of a transition by splitting it and inserting a new state in the middle. One of the two transitions is labeled ϵ.

We described how the transitions inside $Q \times X$ work, but there is also the state set Q. This set is included to model the final cost function. In every state the automaton has the chance to non-deterministically jump from a state (q, x) to a state q' through reading a if $\delta(q, a) = q'$, depending on $\mu(q')$. For example, if $\mu(q) = x_1 + x_2 + c$, then $(q, x_1) \xrightarrow{a} q'$ and $(q, x_2) \xrightarrow{a} q'$. If $c \neq 0$, we insert a construction that generates c additional accepting runs.

The initial costs ν_0 can be modeled with the help of a construction of ϵ-transitions. □

Closure properties examined in [ADD+11] included reversal. The closure for an if-then-else function was shown as well: For a regular language L and functions f_1, f_2 the if-then-else function is given as

$$w \mapsto \begin{cases} f_1(w) \text{ if } w \in L \\ f_2(w) \text{ if } w \notin L \end{cases} .$$

Finally, a regular look-ahead was considered, which allows a CRA to update the registers also depending on the membership to some regular language of the rest of the input word.

As a decidability result, the paper [ADD+11] included finding a minimum. That is, e.g. given a CRA over the natural numbers and addition, finding the smallest output value that can be generated. Also, for certain algebras, it is decidable whether two CRAs are equivalent. Another problem is dominance: Given two machines, will the first machine always generate a greater value than the second one? The authors also considered searching for a given value, so for a value d of the algebra, the task would be to find out whether there is an input word for which the output is d.

We want to add another decidability problem to this list, which is boundedness. Can the output value be bound by a constant for a given a CRA? This formulation assumes an order on the algebra values, but we can formulate the problem equivalently in the following way: Does the realized function have a finite image? That is, given a CRA $\mathcal{M}$, is $|F(\mathcal{M})(\Sigma^*)| \in \mathbb{N}$? The boundedness property becomes especially interesting if one is to extend the CRA model to infinite inputs like ω-words. An example for such a decidability result is the following:

Theorem 54. *Given a CRA $\mathcal{M}$ over the algebra $\mathcal{A} = (\mathbb{N}; +, \times)$, it is decidable whether $F_{\mathcal{A}}(\mathcal{M})$ is bounded.*

Proof. Let $\Sigma^{\leq n}$ be the set of words up to length n. Consider the image $F_{\mathcal{A}}(\mathcal{M})(\Sigma^{\leq n})$ now where $n = |Q|$ is the set of states. It contains all values obtainable through words that are bound in length by the number of states. All longer words in $\Sigma^* \setminus \Sigma^{\leq n}$ induce a run on $\mathcal{M}$ that has a loop. Now, if we compare $F_{\mathcal{A}}(\mathcal{M})(\Sigma^{\leq n})$ and $F_{\mathcal{A}}(\mathcal{M})(\Sigma^{\leq 2n})$, we get two possibilities: Either both sets are equal; in this case $F_{\mathcal{A}}(\mathcal{M})(\Sigma^{\leq n}) = F_{\mathcal{A}}(\mathcal{M})(\Sigma^{\leq in})$ for all $i \in \mathbb{N}$, hence $F_{\mathcal{A}}(\mathcal{M})(\Sigma^*)$ is finite, and thus $F_{\mathcal{A}}(\mathcal{M})$ is bounded; or the sets are not equal, which means that there exists a loop letting the image grow, hence $F_{\mathcal{A}}(\mathcal{M})$ is not bounded. $\square$

If a CRA over an algebra $\mathcal{A}$ recognizes a bounded cost function, we can find an equivalent CRA over a finite algebra.

Boundedness is a property that, in a way, limits the complexity of the algebra computations a CRA performs. If a cost function of some CRA is bounded, it potentially has positive implications for the complexity upper bounds. Boundedness, however, is a very strong restriction. There are weaker restrictions that, for example, filter out bad examples like the following:

Example 55. *Consider a CRA over the algebra $(\mathbb{N}; +, \times, 0, 1)$ with one state and one register over a one-letter alphabet that has a register update rule that implements the mapping $x \mapsto x \times x$. If the initial value is 2, the realized function is $w \mapsto 2^{2^{|w|}}$. So, the result needs an exponential number of bits to be represented, which results in a high complexity bound.*

To filter out examples like the one above the following restriction is sufficient. For that we first have to extend the definition of linear terms to sets of terms. A linear term set is a set of linear terms in which each variable appears at most in one term in the set.

Definition 56 (Copylessness for CRAs (CCRAs))**.** *A CRA with state set Q, register set X and register update function ρ is called copyless if the set $\{\rho(q, a)(x) \mid x \in X\}$ is a linear term set for each $q \in Q$, $a \in \Sigma$.*

The copylessness restriction is a syntactical one and enforces that the computed term has linear size, i.e. $|V(F_{\mathcal{T}(\sigma(\mathcal{A}))}(\mathcal{M})(w))| \in \mathcal{O}(|w|)$, where $\mathcal{M}$ is the CRA; recall that a term is a tree, i.e. a graph, and $V(G)$ addresses the vertex set of a graph G. Many interesting cases of cost function examples happen to be copyless.

CCRAs have the disadvantage that it lacks certain properties. For example, its functions are not closed under reversal [ADD$^+$11, MR15].

With regard to complexity, copylessness solves the problem that arose in the previous example. In actuality this is stricter than it has to be. A first thing one can do is to change the property to be semantical. That means we only require the term to stay linear. This poses a contrast to copylessness, which also dictates how this goal is achieved; this is similar to determinism and unambiguity for automata. Furthermore, we may even allow polynomial size instead of linear. Put together we gain the following property.

Definition 57 (Polynomially bounded cost function)**.** *The cost function recognized by some CRA $\mathcal{M}$ over an algebra $\mathcal{A}$ is called polynomially bounded if $|V(F_{\mathcal{T}(\sigma(\mathcal{A}))}(\mathcal{M})(w))| \in \mathcal{O}(p(|w|))$ for some polynomial p.*

If $\mathcal{M}$ recognizes a polynomially bounded cost function, we also call $\mathcal{M}$ polynomially bounded. If the polynomial has degree one, we call both the cost function and $\mathcal{M}$ linearly bounded. We directly get the following relationship:

Lemma 58. *Cost functions recognized by CCRAs are linearly bounded.*

Notice that there are indeed CRAs that recognize a cost function that is polynomially bounded but not linearly bounded:

Example 59. *Consider a CRA over the alphabet $\{a\}$ and algebra $(\mathbb{N};+)$ with one state and two registers x and y. We update x by increasing its value in every step by one. The register y is updated by adding the value in x to the old value: $x+y$. As initial values of both registers we choose 0 and $x+y$ is the final output. As a result, this automaton implements the function $a^n \mapsto \frac{n^2+n}{2}$. Written as a term, the result is*

$$\underbrace{1+1+\ldots+1}_{\frac{n^2+n}{2}\ \textit{times}},$$

which is a term of quadratic size.

The output number in the previous example needs a logarithmic number of bits to be represented. Even cost functions that are not polynomially bounded may have an image that only needs a polynomial number of bits.

Finally, we ask the question whether linear boundedness and copylessness actually coincides. Of course, CRAs that are not copyless but recognize a linearly bounded function exist, however, this does not mean that there is no CCRA for such a function.

Theorem 60. *The set of cost functions recognized by CCRAs and the set of cost functions recognized by linearly bounded CRAs, coincide.*

Proof. We already noted that CCRAs recognize linearly bounded functions, so we only have to show the reverse.

Assume some CRA $\mathcal{M}$ over an algebra $\mathcal{A}$ that recognizes a linearly bounded function. Let $c \in \mathbb{N}$ be a constant for which $|V(F_{\tau(\sigma(\mathcal{A}))}(\mathcal{M})(w))| \leq c|w|$ for inputs $w \in \Sigma^*$. Now, $\mathcal{M}$ could contain register updates that make it not copyless, i.e. for a state q, a letter a, and a register x there exist registers x_1 and x_2 such that both $\rho(q,a)(x_1)$ and $\rho(q,a)(x_2)$ are terms that contain x. We will show a procedure to get rid of such situations. From now on let q, a, x, x_1, and x_2 be fixed as described.

If such a copy situation occurs, the problem arises later on when both x_1 and x_2 contribute to the final result term. It is particularly problematic if this happens too often. If x holds a term of linear size, the automaton may only make a constant number of copies contributing to the result; otherwise the result does not stay linear in size.

To resolve said copy situations we first modify the automaton as follows. Let $\approx$ be the equivalence relation on the stats such that $q_1 \approx q_2$ if q_1 and q_2 are in the same strongly connected component (SCC), i.e. if there is a word w_1 for which $q_1 \overset{w_1}{\to} q_2$ and a word w_2 for which $q_2 \overset{w_2}{\to} q_1$. If we generalize the transition relation δ to the equivalence classes defined by $\approx$, we get a DAG. We may assume that this DAG is actually a tree. This can be achieved by duplicating states if necessary.

The first case we look at are SCCs that consist solely of a state q that causes a copy. Let q be such as described above, then we can resolve this copy by duplicating the set of registers. We maintain the set of additional registers like the original ones until the copy in state q occurs at which point we use the original version of x in $\rho(q,a)(x_1)$ and the additional version of x in $\rho(q,a)(x_2)$.

Now, consider a SCC that contains more than one state. In every state q of the SCC every register x could be bounded or not, which means that there exists a finite set of terms V such that for all $w \in \Sigma^*$ with $q_I \overset{w}{\to} q$ it can be guaranteed that after w is read x holds a term in V. If q is a copy state as described, we can resolve the copy tracking the value in x through the states and then directly inserting the right value in $\rho(q,a)(x_1)$ and $\rho(q,a)(x_2)$; x in this case is not used in these two updates any more.

Finally, we are in the situation that copies only occur in SCCs of size greater one and on registers that by hold terms of linear size. Since the state is in a SCC, it is not possible that we make copies of such a register during every run through the loop and then use all copies for the overall result; in this case the resulting term would have at least quadratic size. This means that, if such a copy occurs, only one of the copies can actually end up being part of the overall output.

Let $\mathcal{X}_q \subseteq 2^X$ such that a set of registers is an element of $\mathcal{X}_q$ if from state q it is possible that *all* of them contribute to the overall result. In contrast, if two registers can never contribute to the overall result simultaneously, then there is no subset in $\mathcal{X}_q$ that contains both.

Note that if $x \approx y$, then $\mathcal{X}_x$ and $\mathcal{X}_y$ are equal, disregarding renaming. One can even modify the automaton in a way such that $\mathcal{X}_x = \mathcal{X}_y$ holds. In the automaton we now replace the register set X by $|\mathcal{X}_q|$ many copies. For states that come before the SCC $[q]_\approx$ every duplicate is updated like in the original automaton. This may involve adding some more duplicates of registers to enable the maintenance of the register in question. In the SCC, however, every duplicate is associated with one element of $\mathcal{X}_q$ and only those registers are being updated. This is also true for all states that come ofter $[q]_\approx$.

Now, the state q does not make copies any more. If there was a copy, a set of registers in $\mathcal{X}_q$ would exist in which this copy occurs and all of them would contribute to the overall result. This can happen every time q is reached. We also know that the register in question is not bounded. This means that the overall result would have quadratic size, which is a contradiction.

If we perform the previous construction for all SCCs and register updates, we arrive at a copyless CRAs.

□

Note that in the previous proof the blow-up to simulate a linearly bounded CRA by a copyless one is quite large. It remains an open question whether this blow-up is necessary. If it is, linearly bounded CRAs can act as a more succinct model compared to CCRAs.

7.3.2 Cost Functions as Wreath Products

As we already pointed out, the main characteristic of CRAs is that they perform register updates data-independently. If we look at the semantics, we see that the computation result is obtained by first computing the run through the states and then using this run to compute a term whose evaluation over the given algebra is the output. This two-step computation can be characterized in different ways, one of which is to look at it as a transduction that first computes the run. The initial work on CRAs also stated a definition based on transducers.

There, however, is an equivalent algebraic view, which we want to explore. Recall that the syntactic monoid of a regular language is isomorphic to the transformation monoid of the minimal DFA recognizing the language. In the case of CRAs we do not have final states or a language, but we can still can consider the transformation monoid. Computing the run is equal to computing the image of the natural homomorphism for all prefixes. These values direct how the term is assembled. This mechanism is precisely captured by the *wreath product*.

The wreath product is usually used in terms of semigroups as a way to construct semidirect products. Given two semigoups $(S; \otimes_S)$ and $(T; \otimes_T)$, the wreath product $S \wr T$ is defined as $(S^T \times T; \otimes)$. Its operation is defined as $(f_1, t_2) \otimes (f_2, t_2) = (f_1 \otimes_S (t_1(f_2)), t_1 \otimes_T t_2)$, where $\otimes_S$ is lifted to functions $T \to S$ by $(f \otimes_S g)(t) = f(t) \otimes_S g(t)$. Also, there is an action of T on S^T: For $t, t' \in T$ and $f\colon T \to S$ we define $t(f)$ as $t(f)(t') = f(tt')$.

For the following results we need a wreath product that goes beyond semigroups, that is a wreath product between a register algebra and a semigroup. If we have an arbitrary single-sorted algebra A and a semigroup T, then to define $A \wr T$ we can proceed as in the case of semigroups; we only have to define what $f_1 \otimes_A f_2$ is for $f_1, f_2\colon T \to A$, i.e. we just have to specify which operation of A we choose for $\otimes_A$. In the case of A being a register algebra, we choose $\odot$, which leads to $(f_1, t_1) \otimes (f_2, t_2) = (f_1 \odot (t_1(f_2)), t_1 t_2)$.

Recall that the function $F_{\mathcal{A}}(\mathcal{M})$ associated to a CRA $\mathcal{M}$ is composed of aggregating the register updates, inserting the initial values, applying the final cost function and finally evaluating the resulting term over $\mathcal{A}$. The last three steps will not be our focus now. We want a wreath product characterization for CRAs that captures how the states and registers interact. To that end we define the function $\mathcal{F}'_\sigma(\mathcal{M})\colon \Sigma^* \to R_\sigma^X$ in the following way: $\mathcal{F}'_\sigma(\mathcal{M})(\epsilon)$ is the identity and for $w \in \Sigma^*$,

$$\mathcal{F}'_\sigma(\mathcal{M})(w_1 \dots w_{i-1} w_i) = \mathcal{F}'_\sigma(\mathcal{M})(w_1 \dots w_{i-1}) \odot \rho(\delta^*(q_I, w_1 \dots w_{i-1}), w_i).$$

This function tells us which terms are stored in each register after some input is read. It is straightforward to see that

$$\mathcal{F}_{\mathcal{A}}(\mathcal{M})(w) = \mathrm{eval}_{\mathcal{A}}^{\nu_0}(\mathcal{F}'_\sigma(\mathcal{M})(w) \odot \mu(\delta^*(q_I, w))).$$

The actual computation is therefore captured in $\mathcal{F}'$.

Theorem 61. *For a fixed signature σ and a register set X it holds that for each CRA $\mathcal{M}$ there exists a finite monoid M and a homomorphism*

$$\phi\colon \Sigma^* \to R_\sigma^X \wr M$$

such that

$$\mathcal{F}'_\sigma(\mathcal{M})(w) = \pi_1(\phi(w))(1_M).$$

The reverse also holds, i.e. for each finite monoid M and homomorphism $\phi\colon \Sigma^* \to R^X_\sigma \wr M$ there exists a CRA $\mathcal{M}$ as described such that the equality holds.

Proof. We begin with a CRA $\mathcal{M}$. As the monoid M we choose the transition monoid defined by the transition function of $\mathcal{M}$, so M is a subset of Q^Q for Q being the state set of $\mathcal{M}$. For the transition monoid M, $\eta\colon \Sigma^* \to M$ is the natural homomorphism. We define the homomorphism ϕ as $\phi(a) = (f, \eta(a))$ with $f\colon M \to R^X_\sigma$ defined as $f(m) = \rho(q, a)$ for all $m \in M \subseteq Q^Q$ with $m(q_I) = q$. By an induction over the input word w the correctness of this construction can be verified. If $w = a \in \Sigma$, then $\pi_1(\phi(w))(1_M) = \rho(q_I, a)$. For $w = ua$ we inductively assume that $\pi_1(\phi(u))(\eta(x))$ is equal to $\mathcal{F}'_\sigma(\mathcal{M})(u)$. We now have $\pi_1(\phi(w))(1_M)$, which is equal to $(\pi_1(\phi(u)) \odot \eta(u)\pi_1(\phi(a)))(1_M)$. This again is $\pi_1(\phi(u))(1_M) \odot \pi_1(\phi(a))(\eta(u)) = \mathcal{F}'_\sigma(\mathcal{M})(u) \odot \rho(q, a)$ where $\eta(u)(q_I) = q$. In turn this equals the desired value $\mathcal{F}'_\sigma(\mathcal{M})(ua)$.

For the reverse, a monoid M and homomorphism ϕ are given. We construct a CRA $\mathcal{M}$: We let M be the state set of $\mathcal{M}$ and define the transition function as $\delta(m, a) = m\pi_2(\phi(a))$. The initial state is 1_M. The initial costs ν_0 and the final cost function μ may be arbitrary as they are not part of the property we want to fulfill. For the register update function ρ we choose $\rho(m, a) = \pi_1(\phi(a))(m)$. Again, correctness can be verified by induction.

□

The copylessness property is also captured algebraically and is a straight forward equivalent of the CRA version. The following result can be obtained using the previous proof fitted to the copyless case:

Theorem 62. *For a fixed signature σ and a register set X it holds that for each CCRA $\mathcal{M}$ there exists a finite monoid M and a homomorphism $\phi\colon \Sigma^* \to R^X_\sigma \wr M$ such that*

$$\mathcal{F}'_\sigma(\mathcal{M})(w) = \pi_1(\phi(w))(1_M)$$

and $\{\phi(a)(m)(x) \mid x \in X\}$ is a linear term set for all $a \in \Sigma$ and $m \in M$. The reverse also holds, i.e. for each finite monoid M and homomorphism $\phi\colon \Sigma^* \to R^X_\sigma \wr M$ there exists a CCRA $\mathcal{M}$ as described such that the equality holds.

7.3.3 Cost Register Automata for Well-matched Words

A natural continuation of the previous models can be achieved by going beyond words. In the context of this work, natural candidates would be ranked and unranked trees, nested words and well-matched words. Since all models are very closely related we will not exercise a full framework description for all of them. Instead we follow [KLL16] and only cover the well-matched word case in formal detail.

In the word case an automaton receives register values from the previous step and calculates new values depending on the state. Well-matched words resemble trees, and therefore we have both a horizontal and a vertical dimension. This leads to the following idea for cost register VPAs, which has commonalities with determinized VPAs: The register values in some position have a scope that is the largest well-matched factor that precedes the position. Let w_1 and w_2 be two well-matched words and $f_1 \colon X \to \mathbb{T}^X(\sigma)$ and $f_2 \colon X \to \mathbb{T}^X(\sigma)$ be the two valuations corresponding to w_1 and w_2. Then we get the valuation for w_1w_2 by concatenating f_1 and f_2 just as in the word case. This describes the horizontal component. For the vertical component, we have to state how awb is computed for $a \in \Sigma_{\text{call}}$, $b \in \Sigma_{\text{ret}}$ and w being well-matched, where inductively we already have the valuation f for w. If we regard this situation algebraically, we have w and apply a unary extend operation to it. So, what we do is assigning a term to each extend operation, i.e. each pair of $\Sigma_{\text{call}} \times \Sigma_{\text{ret}}$. These terms must contain variables that take the result of w as well as variables for the valuation that comes before awb. This step is performed when the return letter b is read. Equivalently we can think of these automata as ones that store register values onto the stack. That way they become accessible again when the matching return letter is read. This idea is implemented by the following definition:

Definition 63 (Cost register VPA (CVPA))**.** *A cost register visibly pushdown automaton $\mathcal{M}$ over a single-sorted algebra $\mathcal{A} = (\mathbb{D}; O)$ of signature σ is a tuple*

$$(Q, q_I, \hat{\Sigma}, \Gamma, \delta_{\text{call}}, \delta_{\text{ret}}, \delta_{\text{int}}, X, \nu_0, \rho_{\text{call}}, \rho_{\text{ret}}, \rho_{\text{int}}, \mu),$$

where:

- *Q is the finite set of states.*
- *$q_I \in Q$ is the initial state.*
- *$\hat{\Sigma} = (\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \Sigma_{\text{int}})$ is the visible input alphabet.*
- *Γ is the pushdown alphabet.*
- *$\delta_{\text{call}} \colon Q \times \Sigma_{\text{call}} \to Q \times \Gamma$ is the transition function for call letters.*
- *$\delta_{\text{ret}} \colon Q \times \Sigma_{\text{ret}} \times \Gamma \to Q$ is the transition function for return letters.*
- *$\delta_{\text{int}} \colon Q \times \Sigma_{\text{int}} \to Q$ is the transition function for internal letters.*
- *X is the finite set of registers.*

- $\nu_0 \colon X \to \mathbb{D}$ *is the initial register valuation.*
- $\rho_{\text{call}} \colon Q \times \Sigma_{\text{call}} \to (\mathbb{T}^X(\sigma))^X$ *is the register update function for call letters.*
- $\rho_{\text{ret}} \colon Q \times \Sigma_{\text{ret}} \times \Gamma \to (\mathbb{T}^{X \cup X_{\text{match}}}(\sigma))^X$ *is the register update function for return letters where X_{match} is a copy of X. The copy of $x \in X$ is x_{match}.*
- $\rho_{\text{int}} \colon Q \times \Sigma_{\text{int}} \to (\mathbb{T}^X(\sigma))^X$ *is the register update function for internal letters.*
- $\mu \colon Q \to \mathbb{T}^X(\sigma)$ *is the final cost function.*

By ρ we address the union of ρ_{call}, ρ_{ret}, and ρ_{int}.

A CVPA $\mathcal{M}$ implements a function $F_{\mathcal{A}}(\mathcal{M}) \colon \mathrm{WM}(\hat{\Sigma}) \to \mathbb{D}$. Let $q_0 \dots q_{|w|} \in Q^*$ be the run of $\mathcal{M}$ on an input $w \in \Sigma^*$ where $q_0 = q_I$ is the initial state and let $\gamma_0 \dots \gamma_{|w|} \in \Gamma^*$ be a word defined by the property that γ_i is the letter that is on top of the stack if $w_1 \dots w_i$ is read. We assign each step a valuation of the registers. The initial valuation is $\nu_0 \colon X \to \mathbb{D}$. Assuming ν_j is already computed for all $1 \le j < i$, then $\nu_i(x)$ is defined depending on w_i.

- If w_i is a call or an internal letter, then $\nu_i(x) = \mathrm{eval}_{\mathcal{A}}^{\nu_{i-1}}(\rho(q_{i-1}, w_i)(x))$.
- If w_i is a return letter, then let j be the matching position of i and $\nu_i(x) = \mathrm{eval}_{\mathcal{A}}^{\nu_{i-1},\nu_{j-1}}(\rho_{\text{ret}}(q_{i-1}, w_i, \gamma_i)(x))$. Note that $\rho_{\text{ret}}(q_{i-1}, w_i, \gamma_i)(x) \in \mathbb{T}^{X \cup X_{\text{match}}}(\sigma(\mathcal{A}))$, which means that this term has two sets of variables X and X_{match}. The registers of X we replace by the values in ν_{i-1} and the registers of X_{match} by the values in ν_{j-1}.

After w is read we arrive at a valuation $\nu_{|w|}$. The final output $F_{\mathcal{A}}(\mathcal{M})(w)$ is $\mathrm{eval}_{\mathcal{A}}^{\nu_{|w|}}(\mu(q_{|w|}))$.

Like for CRAs we give a definition for copyless machines. Here, in addition to the ordinary copylessness requirement introduced for CRAs, we essentially have to pay attention to return letters because in the case of copyless CVPA we may also access register values from the matching position, which may result in using register values more than once. Again, another perspective is to regard at the automaton as one that, when reading a call letter, may either use a value in the next step or store it onto the stack.

Definition 64 (Copyless CVPA (CCVPA))**.** *A CVPA with state set Q, register set X and register update functions ρ_{call}, ρ_{ret}, and ρ_{int} is called copyless if the following sets are linear term sets:*

- $\{\rho_{\text{int}}(q, c)(x) \mid x \in X\}$ *for all $q \in Q$ and $c \in \Sigma_{\text{ret}}$.*

- $\{\rho'_{\text{ret}}(q, b, \gamma)(x) \mid x \in X\}$ *for all* $q \in Q$, $\gamma \in \Gamma$, *and* $c \in \Sigma_{\text{ret}}$. *Hereby* ρ_{ret} *stripped of variables in the terms that access register values from the matching position is defined as* ρ'_{ret}.

- $\{\rho_{\text{call}}(q_1, a)(x) \mid x \in X\} \cup \{\rho''_{\text{ret}}(q_2, b, \gamma)(x) \mid x \in X\}$ *for all* $q_1, q_2 \in Q$, $a \in \Sigma_{\text{call}}$, $b \in \Sigma_{\text{ret}}$ *and* $\delta_{\text{call}}(q_1, b) = (q', \gamma)$ *for some* $q' \in Q$. *Here,* ρ''_{ret} *is defined as* ρ_{ret} *being stripped of variables in the terms that access register values from the previous position.*

Note that ρ'_{ret} and ρ''_{ret} are not tightly defined; one possibility is to replace the variables of X, or X_{match} respectively, by some constant.

For the previous definition it is helpful to think of automata for which $\pi_2 \circ \delta_{\text{call}}$ is injective, i.e. γ determines a and q_1. Then γ also determines where a register value may be used: On the next or on the matching position.

The definition of boundedness directly carries over to the cost functions of CVPAs. Also for specific algebras, deciding this property can be easy; e.g. Theorem 54 directly carries over directly. The definition of polynomial boundedness carries over to CVPAs as well. Again, CCVPAs are a special case of polynomially bounded CVPAs. Further, notice that CVPAs, of course, generalize both VPAs and CRAs.

Theorem 65. *The set of cost functions recognized by CCVPAs and the set of cost functions recognized by CVPAs that are linearly bounded, coincide.*

Proof. This result can be proved in a similar way as Theorem 60. Where in the CRA case a copy situation involves a register x that is used in $\rho(q, a)(x_1)$ and $\rho(q, a)(x_2)$ we have to consider two possibilities for VPAs: x may be used more than once in register updates for the next position or, additionally, for the matching position.

We again define an equivalence relation in the states and perform the analogous modifications. Two states q_1 and q_2 are in relation $q_1 \approx q_2$ if it is possible to alternate between q_1 and q_2 arbitrarily often, i.e. for each $n \in \mathbb{N}$ there exists a word that induces a run that is in $Q^*(q_1 Q^* q_2 Q^*)^n$.

As in the proof of Theorem 60 we first consider equivalence classes of single states. If a copy is made in one of them, we fix this occurrence by duplicating the register set.

Also, similar to the proof of Theorem 60, we consider registers with respect to a equivalence class that is bounded. Again, we use the states to track the values and insert them directly in the update function.

Finally, we consider $\mathcal{X}_q \subseteq 2^X$ for the class $[q]_\approx$ and perform the same construction as in the proof of Theorem 60. By the same argument we maintain correctness. $\square$

One motivation for scrutinizing cost register machines is that they are term-generating devices, but they are also term-evaluating devices if the algebra is not free. This is indicated by the following example. In it, we show that there exists a CVPA that reads a word, which codes an arithmetic formula and has its evaluation as the output. This classical problem is known to be Gap$\mathbf{NC}^1$-hard for the chosen algebra.

Example 66. *Arithmetic formulas over* $(\mathbb{Z};+,\times,0,1)$ *can be evaluated by a CVPA over the same algebra. We code the terms as words in the usual way using parentheses. For the rest of the example we assume for convenience that the formulas are maximally parenthesized, i.e. we have* $(1+(1+1))$ *instead of* $(1+1+1)$. *We can understand words that are formulas as well-matched words over the alphabet* $\{-1,1,[,],+,\times\}$, *where* $[$ *is a push letter,* $]$ *a pop letter, and* $-1,1,+,$ *and* $\times$ *are internal letters. We use* $[$ *and* $]$ *instead of* $($ *and* $)$ *for better readability.*

We now give a formal description of a CVPA

$$\mathcal{M}=(Q,q_I,\hat{\Sigma},\Gamma,\delta_{\text{call}},\delta_{\text{ret}},\delta_{\text{int}},X,\nu_0,\rho_{\text{call}},\rho_{\text{ret}},\rho_{\text{int}},\mu)$$

evaluating formulas. The states are $Q=\{q_I,q_+,q_\times\}$ *and as the stack alphabet we introduce the symbols* P, T *and* I, *which are used to indicate on which state the push happened. The transition function is as follows:*

- *Transitions from* q_I: $\delta_{\text{call}}(q_I,[)=(q_I,I)$, $\delta_{\text{int}}(q_I,1)=\delta_{\text{int}}(q_I,-1)=\delta_{\text{ret}}(q_I,],I)=\delta_{\text{ret}}(q_I,],P)=\delta_{\text{ret}}(q_I,],T)=q_I$, $\delta_{\text{int}}(q_I,+)=q_+$, $\delta_{\text{int}}(q_I,\times)=q_\times$.

- *Transitions from* q_+: $\delta_{\text{call}}(q_+,[)=(q_I,P)$, $\delta_{\text{int}}(q_+,1)=\delta_{\text{int}}(q_+,-1)=q_I$.

- *Transitions from* $q_\times$: $\delta_{\text{call}}(q_\times,[)=(q_I,T)$, $\delta_{\text{int}}(q_\times,1)=\delta_{\text{int}}(q_\times,-1)=q_I$.

There is only one variable $x\in X$. *The initial value for* x *is* 0, *so* $\nu_0(x)=0$. *The register update function can be described as follows:* $\rho_{\text{call}}(q_I,[)(x)=0$, $\rho_{\text{int}}(q_I,1)(x)=1$, $\rho_{\text{int}}(q_I,-1)(x)=-1$, $\rho_{\text{int}}(q_I,+)(x)=\rho_{\text{int}}(q_I,\times)(x)=x$ *and* $\rho_{\text{ret}}(q_I,],I)(x)=x$, $\rho_{\text{ret}}(q_I,],P)(x)=x+x_{\text{match}}$ *and* $\rho_{\text{ret}}(q_I,],T,x)=x\times x_{\text{match}}$. *The other states are as follows:* $\rho_{\text{call}}(q_+,[)(x)=\rho_{\text{call}}(q_\times,[)(x)=0$, $\rho_{\text{int}}(q_+,1)(x)=x+1$, $\rho_{\text{int}}(q_+,-1)(x)=x+(-1)$, $\rho_{\text{int}}(q_\times,1)(x)=x\times 1$, $\rho_{\text{int}}(q_\times,-1)(x)=x\times(-1)$. *Finally, we let* $\mu(q_I)=x$.

We see that the construction in the example actually leads to a copyless CVPA. As a result we can observe that CCVPAs over $(\mathbb{Z};+,\times,0,1)$ recognize Gap$\mathbf{NC}^1$-hard functions. The construction actually works for every algebra, so the evaluation function $\text{eval}_{\mathcal{A}}$ can be represented by CCVPAs. It is not surprising that the automata for this are copyless, or, equivalently, linearly bounded. The automaton reads the input term as a word and after reading, this term is held in the register x.

Similarly to CRAs, we will show that CVPAs indeed generalize counting and the weighted automaton model. Unfortunately we do not get a tight correspondence for counting as in the CRA case.

Theorem 67. *Functions in* #VPA *can be implemented as CVPAs over* $(\mathbb{N}; +, \times)$.

Proof. We combine the ideas of the proof of Theorem 53 with the determinization procedure of VPAs. As mentioned before, the result of a determinization can serve as a normal form.

Given a non-deterministic VPA M with the transition relation δ, we construct a CVPA N that extends the normal form of the determinized automaton to natural numbers. The determinized automaton has a state set $2^{Q\times Q}$ but what we actually need to store is the number of runs, so the information is an element in $\mathbb{N}^{Q\times Q}$. To maintain this information we cannot use the states, but we may use $|Q \times Q|$ registers.

The only thing we have to do now is to shift from specifying the transition function of the determinized VPA, as it is done in the determinization algorithm, to specifying the register update function in a similar way. The underlying automaton of N can be fixed by altering it in such a way that it only stores the call letters read onto the stack. The states are uninteresting as we outsource the computation to the registers, so we only have the initial state q_I. The register update function is defined as follows:

- If N reads a call letter a, all registers (p, p) are set to 1, hence $\rho_{\text{call}}(q_I, a)((p, p)) = 1$ and $\rho_{\text{call}}(q_I, a)((p, q)) = 0$ for $p \neq q$.
- If N reads an internal letter c, then $\rho_{\text{int}}(q_I, c)((p, q)) = \sum_{q'\in\delta(q,c)}(p, q')$.
- If N reads a return letter b and a is the matching call letter then a is now on top of the stack. Then $\rho_{\text{ret}}(q_I, b, a)((p, q)) = \sum_{r\in Q}(p, r)_{\text{match}} \times \sum_{A}(p', q')$, where A is the set of state pairs (p', q') that satisfy that there exists $\gamma \in \Gamma$ such that $(p', \gamma) \in \delta(r, a)$ and $q \in \delta(q', b, \gamma)$.

Let F be the set of final states of M. Then we set $\mu(q_I) = \sum_{q\in I, f\in F}(q, f)$.

Correctness can be verified easily by an induction over the structure of the input. If w is a well-matched input word, then our construction is also correct for awb, where a is a call and b a return letter. Moreover, if our construction is correct on two well-matched w_1 and w_2, then it is also correct on w_1w_2 and aw_1b. □

Theorem 68. *The set of functions realized by weighted VPAs over a semiring $R = (\mathbb{D}; +, \times)$ can be implemented by CVPAs over R.*

Proof. This proof follows the same lines as the previous one and use the idea of determinized VPAs. We again have a register set $Q \times Q$. In an input position each register (q, q') stores the product over paths from q to q' for the largest preceding well-matched factor. As in the previous proof, this information can be maintained in each transition step.

□

7.4 Conclusion

Summary

In this chapter we moved beyond language recognition. Instead of only computing one output bit, we output richer information, a natural number, for example. The first two examples were counting accepting computations in non-deterministic machines and weighted automata. Both, as we have shown, are generalized by the more recent model of cost register automata. We considered cost register automata for words as well as for well-matched words.

The cost register framework is of special interest within the present work, whose main theme is term evaluation. Cost register automata can either be seen as term generating or as term evaluating devices. For example, given an arbitrary algebra $\mathcal{A}$, the evaluation function $\text{eval}_{\mathcal{A}}$ can be implemented by CVPAs over $\mathcal{A}$ if we assume the terms to be appropriately represented by a parenthesized word. Naturally, we are also interested in the complexity of different cost register models although that is the topic of a later chapter.

Since the cost register framework deserves attention in its own right, we initiated research on some structural properties. We showed an algebraic representation of cost functions over words in terms of a wreath product.

To chart the set of cost functions, we also looked at ways to restrict them, the first restriction being the copylessness restriction. While being useful, it turned out that there might be better choices; some being more and some being less restrictive. We proposed the less restrictive property of polynomial boundedness, which has the potential to be beneficial in complexity considerations.

Contributions

In [KLL16] we introduced CVPAs and researched basic properties like closure properties, copylessness, and complexity. The connection to counting was also drawn. In [KLL17a, KLL17b] we revisited the complexity problems.

In addition to the contents of those papers there is more new content. The wreath product representation of functions recognized by CRAs is one. It underlines the central property that makes CRAs tame, which is the one-way flow of information from the states to the register actions.

Also, we generalized the notion of copylessness, which was introduced to obtain a well behaving model. Functions of non-copyless machines might have images of exponential size, which is not desirable for different reasons, complexity reasons being one of them. Copylessness forces the image to be linear in size. Based on this observation we introduced polynomially bounded functions and automata (CRAs and CVPAs) and showed that linearly bounded functions coincide with functions of copyless machines.

Sources and Related Work

Counting is predominantly present in complexity theory. Counting in NFAs has connections to branching programs [CMTV98]. Counting in VPAs has first been considered in [KLM12].

We only briefly touched on the topic of weighted automata as they are a preliminary model for cost register automata. There is a huge body of research for various different variants of this automaton model; see e.g. the survey [GM18].

The cost function framework originates in [ADD$^+$13] as regular cost functions. This work also contains copylessness. Based on this work [AM15] and [AKM17] investigated complexity-related questions for CRA. Complexity questions for CVPAs have been addressed in [KLL16] and [KLL17a, KLL17b]. Equipping VPAs with cost registers was done in [KLL16]. There has also been an approach to extend CRAs by non-determinism [CKL15]. The setting of ranked trees was analyzed in [CL10] and [BCK$^+$14] and a connection to ω-words has been drawn in [CF16]. Logic and algebra-based characterizations of regular cost functions can be found in [Col13, Col17, Col09]. It is worth noting that in these works cost functions are abstracted by a relation that preserves boundedness but forgets concrete values. In [MR16, MR15] a subset of copyless cost functions was proposed, which is supposed to have nicer properties as well as a natural logic characterization.

Further Research

Cost register automata are an active field of research. In the scope of this chapter we were only able to scratch the surface of the topic. There are many different directions to look into next:

- We only defined CVPAs, but it is, of course, also possible to define cost register variants for nested word automata or unranked tree automata as well. It would be nice to see that these two and CVPAs indeed capture functions that correspond tightly to those of CVPAs. This could be achieved by showing that there exist equivalent wreath product characterizations for all of them. We also expect that properties like copylessness and polynomial boundedness carry over.

- It would be interesting to see what can be achieved by using the wreath product representation. One question could be, what impact the monoid in the product has. For instance, what do we know if it is aperiodic? Also, we only considered the wreath product for CRAs; it should be possible to come up with a wreath product of a register algebra and a forest algebra that captures CVPA functions.

- On the other hand, the wreath product characterization can serve as a starting point to come up with a natural generalization of CRAs that should still be tame. Up until now the wreath product is built between a register algebra R and a finite monoid M. The finite monoid dictates what happens in the registers. We could now add another layer of registers R', possibly over a different algebra that is directed by R. For that to be tame, R should be partitioned into a finite set of equivalence classes, possibly just by a finite congruence relation. Maybe typed algebras [BKR11] could also be applied there. Of course, we do not need to stop here; an arbitrary number of register layers would be possible. The tameness stems from the fact that the flow of information only goes from top to bottom and from the past to the future.

- Related to the previous question is whether there exists something like a syntactic algebra to recognize cost functions. This probably needs a notion of minimality of CRAs.

- The initial paper [ADD$^+$11] that introduced CRAs showed closure under regular look-ahead. That means that if a CRA knows whether the rest of the word belongs to some regular set it does not become more powerful. Algebraically, regular look-ahead could be modeled by generalizing the wreath product characterization to a block product.

- Polynomial boundedness seems to be a promising property. As we will show later, this property behaves well with respect to complexity. It would be worth investigating what other applications there are for this notion.
- In general it is undecidable whether a CRA or CVPA is bounded. Polynomial boundedness on the other hand could be decidable; at least if the degree of the polynomial is given. If we had that, we could get a way to decide whether a CRA can be made copyless.
- Every CRA that is copyless is automatically linearly bounded, but it could be possible to have a much smaller automaton for the same function, which is allowed to be not copyless. Hence, an analysis of the blow-up would be another research target.
- There should be many applications for CVPAs in practice. Potentially, applications that involve CRAs can profit from CVPAs. In particular in the area of verification and model checking applications can be seen on the horizon.
- There exists another kind of quantitative automaton model, that is automata over infinite alphabets. There are different models, but one recent example could be integrated into the cost register framework seamlessly: In [DA14] symbolic visibly pushdown automata have been introduced. Here, each input comes from an infinite domain like $\mathbb{N}$; the states are being updated depending on an equivalence relation over $\mathbb{N}$ of finite index. The mentioned idea of an iteration of wreath products already incorporates a similar idea.

Chapter 8

Circuits

As the final computational model we scrutinize circuits. As mentioned before, they are an important modeling tool in low complexity contexts in contrast to Turing machines for higher complexity. A circuit uses wires and gates to compute some value from inputs that are fed by input gates. In that they are very similar to terms. Terms, as we defined them, are trees that can be evaluated over some algebra. Circuits generalize terms with variables in the sense that they do not have to be a tree but just a DAG. So, one could first define circuits and then terms as a special kind of circuit.

In the classical setting, circuits are Boolean, which means they get Boolean values as inputs, the wires transport Boolean values and the gates perform Boolean operations like conjunction or negation. The output then is also a Boolean value. There are also arithmetic circuits, which work with natural numbers or integers, which is a more general model than Boolean circuits. We, however, need even more generality and define circuits in a way that we can utilize any many-sorted algebra as an underlying algebra.

Later we will need circuits that are Boolean, but also allow for computations in some algebra $\mathcal{A}$. Such a circuit then has wires that transport Boolean values and wires that transport values from $\mathcal{A}$. Both kinds of values interact via multiplexer gates. As we will see later, this interaction is implemented by composing the algebra $\mathcal{A}$ with the Boolean algebra $\mathcal{B}$ into a new algebra. So, we will just consider circuits of many-sorted signatures and later plug in such a composed algebra.

Definition 69 (Many-sorted circuit)**.** *Given a signature σ of S sorts, then a many-sorted circuit over signature σ, n inputs and m outputs is a tuple*

$$C = (V, E, \mathsf{Order}, \mathsf{Gatetype}, \mathsf{Outputgates}),$$

where:

- $(V; E)$ *is a directed acyclic graph. Elements of* V *we call gates and elements of* E *we call wires.*
- $\mathsf{Order}\colon E \hookrightarrow \mathbb{N}$ *is an injective map giving an order on the edges.*
- $\mathsf{Gatetype}\colon V \to [|\sigma|] \cup \{x_1, \ldots, x_n\}$ *assigns a position of the signature or makes it an input gate, where* $\{x_1, \ldots, x_n\}$ *are symbols for the* n *inputs.*
- $\mathsf{Outputgates}\colon \{y_1, \ldots, y_m\} \to V$ *promotes gates to output gates.*

Further the following must hold:

- *If some* $v \in V$ *has in-degree* 0*, then* $\mathsf{Gatetype}(v) \in \{x_1, \ldots, x_n\}$ *or* $\mathsf{Ar}_\sigma(\mathsf{Gatetype}(v)) = 0$*.*
- *If some* $v \in V$ *has in-degree greater* 0*, then the in-degree is* $\mathsf{Ar}_\sigma(\mathsf{Gatetype}(v))$*.*
- *For all* $v \in V$*, let* $v_1, \ldots v_{\mathsf{Ar}_\sigma(\mathsf{Gatetype}(v))}$ *be the input gates for* v *such that* $\mathsf{Order}((v_i, v)) \le \mathsf{Order}((v_j, v))$ *if and only if* $i \le j$*. Then* $\mathsf{Out}_\sigma(\mathsf{Gatetype}(v_i)) = \mathsf{In}_\sigma(\mathsf{Gatetype}(v), i)$*. By* $\mathsf{In}_C \subseteq [S]^n$ *we denote a word that holds the sorts of the input gates: If* v_i *is an input gate, then* $\mathsf{In}_C(j) = \mathsf{In}_\sigma(\mathsf{Gatetype}(v_i))$ *where* $\mathsf{Gatetype}(v_i) = x_j$*. Similarly,* $\mathsf{Out}_C \subseteq [S]^m$ *stores the sorts of the output gates.*

By $\mathsf{Circ}_{\sigma,n,m}$ *we denote the set of circuits over* σ *of* n *inputs and* m *outputs.*

Just like terms, circuits are based on some signature. Given a valuation for the input gates and an algebra to be interpreted over, we can evaluate a circuit:

Definition 70 (Evaluation of many-sorted circuits)**.** *Given a signature* σ *of* S *sorts, an algebra* $\mathcal{A} = (\mathbb{D}; O)$ *of signature* σ *with* $\mathbb{D} = (\mathbb{D}_i)_{i\in[S]}$ *and* $O = (\circledast_i)_{i\in[k]}$*, the evaluation map* $\mathrm{eval}_\mathcal{A}\colon \mathsf{Circ}_{\sigma,0,m} \to \mathbb{D}^m$ *is defined on circuits without inputs. For a gate* v *in a circuit* $C \in \mathsf{Circ}_{\sigma,0,m}$*, by* $S(v)$ *we denote the subcircuit of* C *that consists of all gates from which* v *is reachable. Let* $v_1, \ldots v_{\mathsf{Ar}_\sigma(\mathsf{Gatetype}(v))}$ *be the predecessors of* v *ordered by the wire order. Then*

$$\mathrm{eval}_\mathcal{A}(S(v)) = \circledast_{\mathsf{Gatetype}(v)}(\mathrm{eval}_\mathcal{A}(S(v_1)), \ldots, \mathrm{eval}_\mathcal{A}(S(v_{\mathsf{Ar}_\sigma(\mathsf{Gatetype}(v))}))).$$

Let $y_1, \ldots y_m$ *be the output gates, then*

$$\mathrm{eval}_\mathcal{A}(C) = (\mathrm{eval}_\mathcal{A}(S(y_1)), \ldots, \mathrm{eval}_\mathcal{A}(S(y_m))).$$

Given a circuit C *and a fitting input word* $w \in \mathbb{D}^n$*, we let* $\mathrm{eval}_\mathcal{A}\colon \mathsf{Circ}_{\sigma,n,m} \times \mathbb{D}^n \to \mathbb{D}^m$ *with* $\mathrm{eval}_\mathcal{A}(C, w) = \mathrm{eval}_\mathcal{A}(C_w)$*, where* C_w *is the circuit* C *in which the input gates are replaced by constant operations according to* w*.*

Note that according to this definition, inputs can only be values that appear as constant operations in the algebra. When required we relax this and allow arbitrary input values of the domain without running into any trouble.

One circuit of $\mathsf{Circ}_{\sigma,n,m}$ represents a function $\mathbb{D}^n \to \mathbb{D}^m$ for an algebra domain $\mathbb{D}$. In the case of language recognition we have $\mathbb{D} = \mathbb{B}$ and $m = 1$, but note that one circuit only accepts inputs of a fixed length. For arbitrary inputs we need families of circuits $(C_n)_{n\in\mathbb{N}}$ where C_n is a circuit of n inputs. The function one circuit C implements over some algebra $\mathcal{A}$ of the same signature we address by $F_{\mathcal{A}}(C)$ that is defined as $w \mapsto \mathrm{eval}_{\mathcal{A}}(C, w)$ for $w \in \mathbb{B}^n$. If $C = (C_n)_{n\in\mathbb{N}}$ is a family of circuits, we can extend $F_{\mathcal{A}}(C)$ and set $w \mapsto \mathrm{eval}_{\mathcal{A}}(C_{|w|}, w)$. The same goes for the case that $\mathcal{A}$ is a family of algebras. Then the family of algebras and the family of circuits have to have the same family of signatures and we set $F_{\mathcal{A}}(C)$ as $w \mapsto \mathrm{eval}_{\mathcal{A}_{|w|}}(C_{|w|}, w)$. In the case of language recognition we also write $L(C)$ to address the recognized language if the algebra is clear from the context.

Families of circuits in general are very powerful. A family containing only trivial circuits may recognize a unary encoding of the halting problem. This is undesired, so one restricts families by the constraint that there is a way to compute the circuit for input length n within some complexity bound. This is called uniformity. See e.g. [Str94] or [Vol99] for basics in circuit complexity.

8.1 Boolean Circuits

When speaking about Boolean circuits we mean circuits that are evaluated over an algebra with domain $\mathbb{B}$. First, notice that we are not restricted to $\mathbb{B}$ as the input alphabet. An alphabet of k letters can be implemented by assigning each input position k input gates. The i-th input gate of word position j is assigned 1 if and only if the j-th letter of the input word is the i-th letter. From now on we just assume $\mathbb{B}$ as being the input alphabet.

We mainly consider three kinds of gates in the Boolean case: Boolean operations, modulo gates and threshold gates. Besides the type of gates used, the size, depth, and the fan-in of gates are measures we use to define complexity classes.

The class $\mathbf{NC}^i$ consists of all languages that are recognized by a family of circuits that have polynomial size, $\mathcal{O}(\log^i n)$ depth and that are interpreted over the algebra $(\mathbb{B}; \wedge, \vee, \neg, \bot, \top)$. Note that $\wedge$ and $\neg$ already form a base, so we may also use other Boolean operations as they can be simulated. Gates in such circuits have a fan-in of at most two.

The class $\mathbf{NC}^1$ is a very prominent one and also plays a special role in the present work. It is known that for instance regular languages and even visibly pushdown languages can be recognized by $\mathbf{NC}^1$ circuit families. We will derive those results ourself in the course of this work. The class $\mathbf{NC}^0$ carries also relevance as the smallest useful complexity class we consider. In this case an output bit can be computed by a constant-size circuit.

A very important property to keep in mind is that $\mathbf{NC}^1$ circuits can be transformed into trees. It is unknown whether this is possible in $\mathbf{NC}^i$ for $i > 1$.

If we allow for unbounded fan-in for $\wedge$ and $\vee$ gates, we get the $\mathbf{AC}^i$ classes. Formally we introduce a family of algebras where for each input length n we have the operations $\wedge_n \colon \mathbb{B}^n \to \mathbb{B}$ and $\vee_n \colon \mathbb{B}^n \to \mathbb{B}$. By using those, we can simulate gates of unbounded but polynomial fan-in. If we only allow the disjunction gates to have unbounded fan-in, then we get the $\mathbf{SAC}^i$ classes. It holds that $\mathbf{NC}^i \subseteq \mathbf{SAC}^i \subseteq \mathbf{AC}^i$. The classes $\mathbf{AC}^0$ and $\mathbf{SAC}^1$ will become important later: The class $\mathbf{AC}^0$ contains the star-free languages as well as the quasiaperiodic languages and $\mathbf{SAC}^1$ the CFLs.

Modulo gates are also used in circuit complexity. Modulo gates can test whether the number of wires feeding into it that hold the value $\top$ is a multiple of k. Circuit families that we evaluate over the family of algebras with the members $(\mathbb{B}; \wedge_n, \vee_n, \neg, \equiv_n^k)$ for $n \in \mathbb{N}$ are the same as $\mathbf{AC}^0$ but include modulo gates. This class we call $\mathbf{ACC}_k^i$. If we allow for arbitrary modulo gates, we get $\mathbf{ACC}^i$.

Finally, threshold gates are gates of unbounded fan-in that output true if sufficiently many inputs are true, which usually means half of the inputs. The resulting class is $\mathbf{TC}^0$.

We set $\mathbf{NC} = \bigcup_{i\in\mathbb{N}} \mathbf{NC}^i$ and the classes $\mathbf{AC}$, $\mathbf{ACC}$, $\mathbf{SAC}$ and $\mathbf{TC}$ are similarly defined. All these sets are equal. We get the following relation among the classes we are most interested in:

$$\mathbf{AC}^0 \subsetneq \mathbf{ACC}^0 \subseteq \mathbf{TC}^0 \subseteq \mathbf{NC}^1 \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{SAC}^1 \subseteq \mathbf{NC} \subseteq \mathbf{P}$$

There are close relationships between logic and circuit complexity. For now we want to point out two: First, $\mathbf{AC}^0$ captures exactly the languages recognized by first-order formulas over words using arbitrary numerical predicates, i.e. $\mathbf{AC}^0 = \mathsf{FO}[\mathsf{arb}]$, and second $\mathbf{TC}^0$ translates to majority logic. Both show that the gates used by the circuit correspond to the quantification used on the logic side.

In the world of circuit complexity there are almost only open questions with one remarkable exception: The parity language is not in $\mathbf{AC}^0$ and thereby separating $\mathbf{AC}^0$ from $\mathbf{ACC}^0$. However, as we will see in the end of this work, the understanding of the relationship of $\mathbf{AC}^0$ and larger classes appears not to go beyond this separating result.

8.2 Arithmetic Circuits

If we consider $\mathbf{NC}^1$, we see that we do not need negation gates to get the full expressibility since they can be pushed to the input gates using DeMorgan's law. Now, $(\mathbb{B}; \vee, \wedge, \bot, \top)$ and $(\mathbb{N}; +, \times, 0, 1)$ have the same signature. If we evaluate $\mathbf{NC}^1$ circuits over $(\mathbb{N}; +, \times, 0, 1)$, we get a so-called arithmetic circuit and in this case the class $\#\mathbf{NC}^1$. We can do the same with other classes like $\mathbf{AC}^i$, $\mathbf{NC}^i$, and $\mathbf{SAC}^i$, and then get $\#\mathbf{AC}^i$, $\#\mathbf{NC}^i$, and $\#\mathbf{SAC}^i$. Another version of arithmetic circuits is based on integers rather than on naturals. Then the resulting classes are Gap$\mathbf{AC}^i$, Gap$\mathbf{NC}^i$, and Gap$\mathbf{SAC}^i$.

Another perspective on arithmetic circuits is to regard them as counting circuits. We already discussed counting in the case of automata. In circuits the equivalent of an accepting computation in automata is a proof tree. Consider a $\mathbf{NC}^1$ circuit that accepts a certain input. Assume this circuit to be a tree. We can find minimal subcircuits containing the output gate that still evaluate to true. It turns out that this notion truly captures counting: There are classes that have both circuit and automata characterizations and both lead to the same set of functions in terms of counting. Also, counting and arithmetic circuits are equivalent.

One can relate sets of functions of the form $\Sigma^* \to \mathbb{N}$ to languages. This can be applied to the previously defined circuit complexity classes; for instance the problem whether $\mathbf{NC}^1 = \#\mathbf{NC}^1$ is still open.

8.3 Generalized Circuits

The way we defined circuits allows us go beyond Boolean and arithmetic circuits. We can combine any circuit and algebra of the same signature. However, later we will need a Boolean circuit enriched by some algebra. In the case of arithmetic circuits it was sufficient to use addition and multiplication gates since it is possible to simulate a Boolean algebra with it. In general, this is not the case.

The way we approach this is to define algebras that are made of several other algebras. The base is the Boolean algebra $\mathcal{B} = (\mathbb{B}; \wedge, \vee, \neg, \bot, \top)$. The algebras we append will interact via multiplexer operations.

Definition 71 (Multiplexer operation)**.** *Given a set X, the ternary multiplexer operation is defined as* $\mathrm{mp}_X \colon \mathbb{B} \times X \times X \to X$ *with*

$$(b, x, y) \mapsto \begin{cases} x & \textit{if } b = \bot \\ y & \textit{else} \end{cases} .$$

This gives us the means to define compositions of algebras.

Definition 72 (Composition of algebras)**.** *Given n possibly many-sorted algebras $\mathcal{A}(i) = (\mathbb{D}(i); O(i))$ for $i \in [n]$, the composition $(\mathcal{A}(i), \dots, \mathcal{A}(n))_{\mathcal{B}}$ of these algebras is the algebra*

$$(\mathbb{B}, \mathbb{D}(1), \dots, \mathbb{D}(n); \wedge, \vee, \neg, \bot, \top, \mathrm{mp}_{\mathbb{D}(1)}, \dots, \mathrm{mp}_{\mathbb{D}(n)}, O(1), \dots, O(n)).$$

The previous definition also naturally carries over to families of algebras. We presume composition to be associative, which means that, for example, we see $((\mathcal{A}_1, \mathcal{A}_2)_{\mathcal{B}}, \mathcal{A}_3)_{\mathcal{B}}$ as $(\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)_{\mathcal{B}}$ although these two are formally not identical. Note that $(\mathcal{A})_{\mathcal{B}}$ is not the same as $\mathcal{A}$.

Now, we can define classes similar to e.g. $\mathbf{NC}^i$ that are enriched by some algebra. Intuitively, the Boolean part is directing the non-Boolean part via multiplexer gates.

Definition 73 ($\mathcal{A}$-$\mathbf{NC}^i$, $\mathcal{A}$-$\mathbf{NC}^i_D$)**.** *For a many-sorted algebra $\mathcal{A} = (\mathbb{D}; O)$ and subdomain D of $\mathcal{A}$, the set $\mathcal{A}$-$\mathbf{NC}^i_D$ contains all functions $F_{(\mathcal{A})_{\mathcal{B}}}(C)$, where C is a family of circuits having the same family of signatures as $(\mathcal{A})_{\mathcal{B}}$ that contains circuits of polynomial size, depth $\mathcal{O}(\log^i n)$, inputs of D and one output of a subdomain of $\mathcal{A}$. For the special case of Boolean inputs we set $\mathcal{A}$-$\mathbf{NC}^i = \mathcal{A}$-$\mathbf{NC}^i_{\mathbb{B}}$.*

Notice that, for example, we have $(\mathbb{N}; +, \times, 0, 1)$-$\mathbf{NC}^1 = \#\mathbf{NC}^1$.

8.4 Conclusion

Summary

Following the gist of the previous chapters, we gave very general definitions for circuits. We use circuits to model functions based on some signature. In that they generalize terms. Given a fitting algebra and an input, a circuit can be evaluated.

We embedded Boolean and arithmetic circuits into this set of definitions. We also defined circuits that combine Boolean circuits and arbitrary algebras by multiplexer gates.

Contributions

The level of generality we offer is non-standard. The generalized circuits are new; they are tailored to our later needs. Early versions of our definitions can be found in [KLL17a, KLL17b].

Sources and Related Work

An overview of circuit complexity can be found in [Vol99]. Arithmetic circuits and counting complexity classes were surveyed in [All04].

Further Research

It would be interesting whether there are more classical classes that could be captured by generalized circuits, i.e. by Boolean circuits augmented with an additional algebra via multiplexer gates. It is worthwhile to investigate relationships to more classical complexity classes.

Part II

Evaluation Complexity

Chapter 9

General Evaluation Complexity

In this chapter we consider the general problem of evaluating terms over an arbitrary algebra $\mathcal{A}$. There is no actual restriction on the algebras but for the construction itself that we will present we presume the algebra to be single-sorted as well as consisting only of constant and binary operations. These restrictions help to keep the presentation concise and it is easy to see that the construction also works for cases lacking these restrictions.

As a result of our construction we get a circuit family that is an enriched $\mathbf{NC}^1$ circuit family, which means that we need to give the circuit the ability to make computations within $\mathcal{A}$. In fact, we need a bit more than the original algebra $\mathcal{A}$. The following algebra precisely captures the operations we need to make our circuit construction work:

Definition 74 (Functional algebra)**.** *Given is an algebra $\mathcal{A} = (\mathbb{D}; B, Z)$ over a single-sorted signature σ, where B contains the binary operations and Z the 0-ary operations. The functional algebra is defined as*

$$\mathcal{F}(\mathcal{A}) = (\mathbb{D}, \widetilde{\mathbb{D}}; B, Z, \widetilde{B}, \circ, \odot, \mathrm{id}),$$

where $\widetilde{\mathbb{D}} \subseteq \mathbb{D}^{\mathbb{D}}$ and the additional operations are as follows:

- $\circ \colon \widetilde{\mathbb{D}} \times \widetilde{\mathbb{D}} \to \widetilde{\mathbb{D}}$ *is the functional composition.*
- $\odot \colon \widetilde{\mathbb{D}} \times \mathbb{D} \to \mathbb{D}$ *is the substitution, i.e. $f \odot c = f(c)$.*
- $\mathrm{id} \in \widetilde{\mathbb{D}}$ *is the identity map, which is a constant operation.*
- *For each binary operation $\circledast \colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ in B there exists an operation $\widetilde{\circledast} \colon \widetilde{\mathbb{D}} \times \mathbb{D} \to \widetilde{\mathbb{D}}$ in $\widetilde{B}$, which is defined as $(f \widetilde{\circledast} c)(x) = f(x) \circledast c$.*

Now, $\widetilde{\mathbb{D}}$ is the set of functions generated by those operations.

This is a two-sorted algebra which, in addition to the domains and operations of $\mathcal{A}$, contains another domain, which consists of functions. These functions can be thought of as linear functions.

Given a term t over $\mathcal{A}$, moving to $\mathcal{F}(\mathcal{A})$ enables us to obtain more ways to generate t. For example, if we have a term $t = (((\ldots(a \circledast a)\ldots) \circledast a) \circledast a)$ that has the shape of a list, we can also represent t as an equivalent term $t' = t_1 \odot t_2$ where the first half of t is a context t_1 and t_2 is the second half. Note that t' only has half the depth of t.

One can also see that the functional algebra shares great similarity to forest algebras. The domain $\mathbb{D}$ can be regarded as the horizontal monoid and $\widetilde{\mathbb{D}}$ as the vertical one. The operation $\circ$ in $\mathcal{F}(\mathcal{A})$ corresponds to $\cdot$ in forest algebras and $\odot$ to $\cdot'$. Further, id corresponds to 1. A main difference is that forest algebras are designed for unranked forests whereas the functional algebras are designed for ranked algebras. That is why the $+$ operation in forest algebras can be found more indirectly in functional algebras. The functional algebras fit precisely the purpose we will use it for whereas forest algebras would be unnecessarily powerful here.

Notice that for binary operations $\circledast\colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ we introduced a functional version $\widetilde{\circledast}\colon \widetilde{\mathbb{D}} \times \mathbb{D} \to \widetilde{\mathbb{D}}$ but no symmetrical version $\mathbb{D} \times \widetilde{\mathbb{D}} \to \mathbb{D}$. It turns out that in our construction we only need the version defined because of the normal form of terms we will choose. If desired, the definition of functional algebra can be easily extended to arbitrary arities. In this case exactly one of the inputs is a function. Also, we omitted unary operations since they can be simulated by a binary one that ignores one of the inputs. Without losing generality, we will only consider algebras that have only 0-ary and binary operations.

Given $\mathcal{A}$, the algebra $\mathcal{F}(\mathcal{A})$ consists of two domains. One could also consider a similar definition where $\mathcal{F}(\mathcal{A})$ only has the domain $\widetilde{\mathbb{D}}$ as $\mathcal{A}$ can be embedded through letting the elements of $\mathbb{D}$ be constant functions. This would lead to a partial algebra. There would be no conceptual problem utilizing this view, but for clarity we do not do it.

If we want to reason about the complexity of term evaluation, we have to address how a term is given as an input for an algorithm. Recall that, given a signature σ, a term t of $\mathbb{T}(\sigma)$ can be represented as a word over the alphabet consisting of parentheses and operation symbols. There is an isomorphism between terms and word representations of terms. Thus, in the following we will simply treat terms as words.

Our main evaluation theorem is the following:

Theorem 75. *Given a possibly many-sorted algebra $\mathcal{A} = (\mathbb{D}; O)$ of signature σ and domain $\mathbb{D}$, the evaluation function* $\mathrm{eval}_{\mathcal{A}} \colon \mathbb{T}(\sigma) \to \mathbb{D}$ *is in DLOGTIME-uniform* $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$.

Note that the input is Boolean but the output is an algebra element. Also, we allowed the algebra to be many-sorted. For the evaluation algorithm it is convenient to assume $\mathbb{D}$ to be the union of subdomains. So, rather than a many-sorted algebra, we would consider a single-sorted partial algebra whereas the partiality of the algebra never plays a role in the constructions since for all well-formed inputs, that is inputs having the right signature, all computations we perform are defined. The union approach works if the subdomains $\mathbb{D}_1, \ldots, \mathbb{D}_S$ are disjoint. Since this may not be given under all circumstances, we choose the following union $\bigcup_{i \in [S]} \mathbb{D}_i \times \{i\}$. Without going into details, we claim that building these cross products does not increase any complexity.

We get a log-depth construction on the expense that we need more complicated algebra computations. A different tradeoff is a naive linear time evaluation where only computations in the original algebra occur.

Notice that the theorem and its proof are independent of the actual algebra $\mathcal{A}$ and thereby strictly syntactical. Later we will consider applications that make use of concrete algebras. Also, it is then when we relate an abstract complexity class like $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$ to ordinary classes known from complexity theory.

9.1 Representing Terms

As mentioned before, we assume all operations used in terms to be either constant or binary. Operations that are ternary or of even higher arity can be split into several binary operations. A construction for this makes use of additional sorts without introducing additional complexity. This can be interpreted as a way of Currying.

Terms written like $(\phi \circledast \psi)$, where ϕ and ψ are also terms, are infix representations. Instead of infix representations, we can also consider postfix representations. If ϕ' and ψ' are the equivalent postfix representations for ϕ and ψ as infix representations, then $\phi'\psi'\circledast$ is the postfix equivalent of $(\phi \circledast \psi)$. Note that conveniently we do not need parentheses any more.

We will make use of the following normal form.

Definition 76 (Postfix normal form)**.** *A postfix term is in postfix normal form (PNF) if for all subterms $\phi\psi\circledast$, ϕ, and ψ holds that $|\phi| \geq |\psi|$.*

In order to convert any term into PNF we need to take care of possible non-commutative operations. To that end we presume all algebras to have symmetric

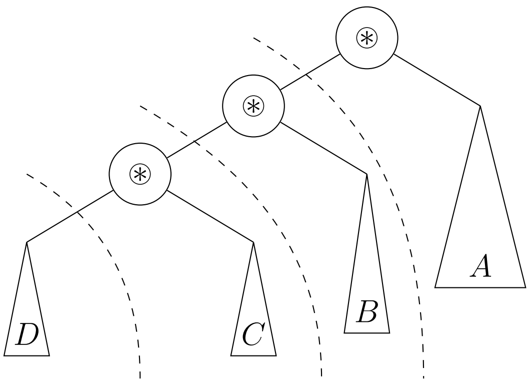


Figure 9.1: The figure shows a PNF term T with the first three left-most operation symbols from the top pointed out. The term T is of the form $DC \circledast B \circledast A\circledast$, where A, B, C, and D are again terms. Note that $|A| \leq |DC \circledast B \circledast|$, $|B| \leq DC\circledast$, and $|C| \leq |D|$. The dashed lines indicate where the term can be split such that the left part corresponds to a closed term. e.g. the middle line gives us the prefix $DC\circledast$, which is again a valid term. What is left on the right side are open terms.

variants of operations, e.g. for an operation $\circledast$ we assume there exists an operation $\circledast'$ in the algebra such that $x \circledast y = y \circledast' x$.

From now on we focus on PNF terms without always explicitly calling it PNF. For the algorithm we put emphasis on the fact that terms are trees coded as words. As of now, we want to distinguish between open and closed terms. The terms defined so far are called closed in opposition to open terms:

Definition 77 (Open term). *We call a word T an open term if there exists a closed non-empty term T' such that $T'T$ is a closed term (in PNF).*

So, open terms are suffixes of closed terms. If we think about the tree that a term represents, then taking a suffix, which is an open term, corresponds to chopping off one of the left-most subtrees; see Figure 9.1. Also, we can concatenate open subterms within a term and obtain an open term again. An open term concatenated with a closed term results again in a closed term. In fact, open terms correspond to contexts $\mathbb{C}(\sigma(\mathcal{A}))$. At this point we get a connection to the functional algebra $\mathcal{F}(\mathcal{A})$ whose second domain are the contexts. So, while closed terms evaluate over $\mathcal{A}$, open terms evaluate over $\mathcal{F}(\mathcal{A})$.

9.2 Dividing Terms

As we already mentioned, $\mathcal{F}(\mathcal{A})$ gives us the means to evaluate a term in more numerous ways, i.e. more numerous ways to split a term so that we can assign meaning to the resulting parts. The main algorithm for term evaluation will work in a recursive manner, which has to carefully decide how to split terms. The main challenge to design a recursive algorithm in $\mathbf{NC}^1$ lies in the fact that we can basically only do static recursion on the input, but the input structure is unknown beforehand. The circuit that implements the recursion has to be some fixed tree where the input may represent any tree, e.g. a balanced tree or a degenerated tree, i.e. a list. A way how to split terms for the recursions is the topic of this section.

We begin with some notation. Given a term T of length n, then for $1 \leq i \leq j \leq n$ we write $i <_T j$ if in the tree j there is an ancestor of i. For convenience, by $[i,j]$ we ambiguously mean the interval $\{i,\dots,j\}$ as well the factor $T_i \dots T_j$ and it will always be clear from the context what we mean. If $i <_T j$ and $[i,j]$ is a term, then we write $i \lhd_T j$.

For the following, let T be a closed PNF term. It ranges from 1 to n. We want to evaluate subintervals $[l,r]$. The size of the interval is $s = r - l + 1$. Such an interval has a middle position. Depending on whether s is even or odd we could define a middle by rounding up or down, but actually we need to be flexible about that and take the middle position as a parameter. So, we are given not only l and r, but also a position m, which is the middle position; it can be $\lfloor l + \frac{s}{2} \rfloor$ or $\lceil l + \frac{s}{2} \rceil$. The two interval borders we will use for the recursion intervals are $l' = \lfloor l + \frac{s}{3} \rfloor$ and $r' = \lceil l + \frac{2s}{3} \rceil$. This divides the interval $[l,r]$ into thirds. We not only consider the three thirds individually, but also the first two thirds as well as the second two thirds together. These five intervals will be our recursion intervals. Based on these static intervals we define some dynamic intervals, i.e. intervals depending on the input. The key later is that these dynamic intervals always lie in one of the five intervals defined previously.

Definition 78 ($\mathcal{M}$, $\mathcal{N}$, $\mathcal{L}$, $\mathcal{R}$, $\mathcal{O}$)**.** *Given $[l,r]$ we define the following subintervals in case they exist; otherwise we define them to be empty:*

- *The largest closed or open subterm in $[l,r]$ that contains m, which is denoted as $\mathcal{M}(l,m,r) = [\mathcal{M}^1(l,m,r), \mathcal{M}^2(l,m,r)]$.*

- *The subinterval in $[l,r]$ that begins with*

$$\max\{p \mid l \leq p \leq m \wedge l-1 <_T p-1 \wedge l-1 \not<_T p\}$$

and ends with the largest position $q \in [m,r]$ such that $[p,q]$ is an open term. This interval is denoted as $\mathcal{N}(l,m,r) = [\mathcal{N}^1(l,m,r), \mathcal{N}^2(l,m,r)]$. The idea here is that, in contrast to $\mathcal{M}(l,m,r)$, which has maximal spread to the left and to the right, $\mathcal{N}(l,m,r)$ has maximal spread to the right and minimal spread to the left.

- *The largest open subterm in $[l,r]$ that precedes $\mathcal{M}(l,m,r)$. This interval is denoted as $\mathcal{L}(l,m,r) = [\mathcal{L}^1(l,m,r), \mathcal{L}^2(l,m,r)]$. It is $\mathcal{L}^2(l,m,r) + 1 = \mathcal{M}^1(l,m,r)$.*

- *The largest open subterm in $[l,r]$ that follows $[\mathcal{M}^1(l,m,r), \mathcal{M}^2(l,m,r) + 1]$. This interval is denoted as $\mathcal{R}(l,m,r) = [\mathcal{R}^1(l,m,r), \mathcal{R}^2(l,m,r)]$. It is $\mathcal{R}^1(l,m,r) - 2 = \mathcal{M}^2(l,m,r)$ and $\mathcal{M}^2(l,m,r)+1$ is a binary operation symbol. We denote the set containing its position by $\mathcal{O}(l,m,r) = \{\mathcal{M}^2(l,m,r) + 1\}$.*

Clearly, the interval is given by (l,m,r), we drop it in the notation and simply write $\mathcal{M}$, $\mathcal{N}$, $\mathcal{L}$, $\mathcal{R}$, and $\mathcal{O}$.

An $\mathcal{M}$-interval could be an open or a closed term. An $\mathcal{N}$-interval by definition always is an open term since later we only need the open ones. The $\mathcal{L}$-interval is also defined to be open, however, even if we allowed it to be closed, it would still be always open because it is shorter than $\mathcal{M} \cup \mathcal{N}$, and so cannot be the complete second operand of the operation in $\mathcal{O}$. In the case of $\mathcal{R}$ we again are only interested in open terms due to the way we use it. Figure 9.2 shows the intervals defined and how they are applied.

The intervals might be empty, however, importantly, they are unambiguous, which is immediately clear for all except for $\mathcal{M}$. This can be seen by maximality and the following lemma:

Lemma 79. *Given the intervals $[p_1, q_1] \subseteq [l, r]$ and $[p_2, q_2] \subseteq [l, r]$, which address closed or open terms, and with the condition that $[p_1, q_1] \cap [p_2, q_2] \neq \emptyset$ then $[p_1, q_1] \cup [p_2, q_2]$ is also a closed or open term.*

Proof. Suppose that $p_1 < p_2 \leq q_1 < q_2$ because otherwise the statement is trivial. The interval $[p_2, q_2]$ has to be an open term since otherwise $p_1 \not<_T q_1$. So, as $p_2 <_T q_1$ holds we have that $[p_2, q_1]$ is an open term, and so $[p_1, p_2 - 1]$ is also a term; it could be open or closed. By combining all parts, we obtain that $[p_1, q_2]$ is a term and it is closed or open depending whether $[p_1, q_1]$ is closed or open. □

The next lemma shows that $\mathcal{M}$ and $\mathcal{N}$ fit together seamlessly.

Lemma 80. *It holds $\mathcal{M}^2 = \mathcal{N}^2$ and $\mathcal{M}^2(l, l', r' - 1) + 1 = \mathcal{N}^1(l' + 1, r', r)$.*

Proof. For the first statement note that $\mathcal{N} \subseteq \mathcal{M}$ and hence $\mathcal{N}^2 \leq \mathcal{M}^2$, which follows from the previous lemma. Now assume that $\mathcal{N}^2$ is strictly smaller than $\mathcal{M}^2$. Let p be the position for which $[p, \mathcal{N}^2]$ is closed. If $p \in \mathcal{M}$, then we find $q \in \mathcal{M}$ and $q \geq \mathcal{N}^2$ such that $[p, q]$ is an open term. But then $[\mathcal{N}^2 + 1, q]$ is also an open term, and so is $\mathcal{N} \cup [\mathcal{N}^2 + 1, q]$. Hence, the maximality of $\mathcal{N}^2$ is violated again. If $p \notin \mathcal{M}$,

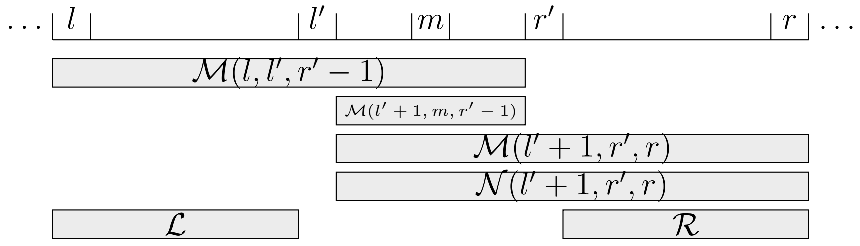


Figure 9.2: The figure shows how a recursion interval is subdivided into smaller recursion intervals. In this case the subdivision for computing $\mathcal{M}(l,m,r)$ is shown. The five intervals recursively yield six values, which may be used to be combined in order to get the evaluation of the $\mathcal{M}(l,m,r)$-interval.

then $[\mathcal{M}^1, \mathcal{N}^1 - 1]$ is an open term, and so is $[\mathcal{M}^1, \mathcal{N}^2]$. But then also $[\mathcal{N}^1, \mathcal{M}^2]$ is an open term and again the maximality of $\mathcal{N}^2$ is violated.

For the second statement, first notice that $\mathcal{M}(l,l',r'-1) \cup \mathcal{N}(l'+1,r',r)$ is indeed an interval, i.e. $[\mathcal{M}^2(l,l',r'-1)+1, \mathcal{N}^1(l'+1,r',r)-1]$ is empty. We get this through the maximality of $\mathcal{M}^2(l,l',r'-1)$. Also $\mathcal{M}(l,l',r'-1) \cup \mathcal{N}(l'+1,r',r)$ is a closed or open term, depending on whether $\mathcal{M}(l,l',r'-1)$ is closed or open. If not empty, the intersection $\mathcal{M}(l,l',r'-1) \cup \mathcal{N}(l'+1,r',r)$ has to be an open term and hence $N^1(l'+1,r',r)$ was not chosen maximal. □

The key lemmas that later constitute the recursive evaluation algorithm are the following ones. They show how to actually compose a term by subterms coming from static subintervals. First, we see how to obtain the $\mathcal{M}$-interval recursively.

Lemma 81. *Given a term T and subinterval $[l,r]$ with middle m, $\mathcal{M}$ equals one of the following intervals:*

1. $\mathcal{M}(l,l',r'-1)$
2. $\mathcal{M}(l'+1,r',r)$
3. $\mathcal{M}(l'+1,m,r'-1)$
4. $\mathcal{M}(l,l',r'-1) \cup \mathcal{N}(l'+1,r',r)$
5. $\mathcal{L}(l,m,r) \cup \mathcal{M}(l,l',r'-1) \cup \mathcal{N}(l'+1,r',r) \cup \mathcal{O}(l,m,r) \cup \mathcal{R}(l,m,r)$

Further the sets involved in the unions of case 4 and 5 are disjoint.

Proof. If $\mathcal{M}$ is entirely contained in $[l,r'-1]$, $[l'+1,r]$ or $[l'+1,r'-1]$, then it coincides with one of the first three cases.

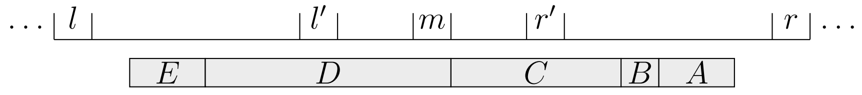


Figure 9.3: In case five for $\mathcal{M}$ as shown in Lemma 81, the interval is subdivided into five parts. We see that DC is a closed term where $D = \mathcal{M}(l, l', r' - 1)$ and $C = \mathcal{N}(l' + 1, r', r)$. Further, $B = \mathcal{O}(l, m, r)$ consists of a single position, which is an operation symbol and $A = \mathcal{R}(l, m, r)$ and $E = \mathcal{L}(l, m, r)$ are open terms.

If the term stretches from the first third to the last third, it is not entirely contained in one of the thirds. Let A be $\mathcal{M}(l, l', r' - 1) \cup \mathcal{N}(l' + 1, r', r)$. By Lemma 80 we know this interval is a disjoint union. Further, A is a closed or open term contained in $\mathcal{M}$ that contains m. If $A = \mathcal{M}$, we are done as case 4 holds.

The interval $\mathcal{M}(l, l', r' - 1)$ is open if and only if A is open, but then $\mathcal{M}^1 = \mathcal{M}^1(l, l', r' - 1)$ due to the minimality of $\mathcal{M}^1(l, l', r' - 1)$. Similarly, it holds that $\mathcal{M}^2 = \mathcal{N}^2(l' + 1, r', r)$. So, we get $A = \mathcal{M}$ and case 4 holds.

Now, suppose that A is a closed term. The term A is part of a larger and possibly open term. It has either the form $AB\circledast$ or $BA\circledast$ where B is a closed term. If $AB\circledast$ is the case, then $\circledast$ lies outside $[l, r]$ and case 4 holds which we again get by a maximality argument. If $BA\circledast$ is the case, then $\mathcal{O}(l, m, r)$ addresses the operation $\circledast$. Let B' be the largest suffix of B that is an open term and a subset of $[l, r]$. Note that B' is a proper suffix because $|B| \geq |A|$ and $|A|$ is more than one third of $r - l + 1$. The interval $\mathcal{L}$ coincides with B'.

The subterm $B'A\circledast = \mathcal{L}(l, m, r) \cup \mathcal{M}(l, l', r' - 1) \cup \mathcal{N}(l' + 1, r', r) \cup \mathcal{O}(l, m, r)$ can be followed by an open term and we get an open term again if we unite those.

The maximal one in $[l, r]$ is addressed by $\mathcal{R}(l, m, r)$. Notice that $\mathcal{L}(l, m, r)$ and $\mathcal{R}(l, m, r)$ might be empty. This concludes the fifth case.

$\square$

Figures 9.3 and 9.4 show how the interval is subdivided in case five.

In a very similar way we can treat $\mathcal{N}$.

Lemma 82. *Given a term T and subinterval $[l, r]$ with middle m, $\mathcal{N}$ equals one of the following intervals:*

1. $\mathcal{N}(l, l', r' - 1)$
2. $\mathcal{N}(l' + 1, r', r)$
3. $\mathcal{N}(l' + 1, m, r' - 1)$
4. $\mathcal{N}(l, l', r' - 1) \cup \mathcal{N}(l' + 1, r', r)$

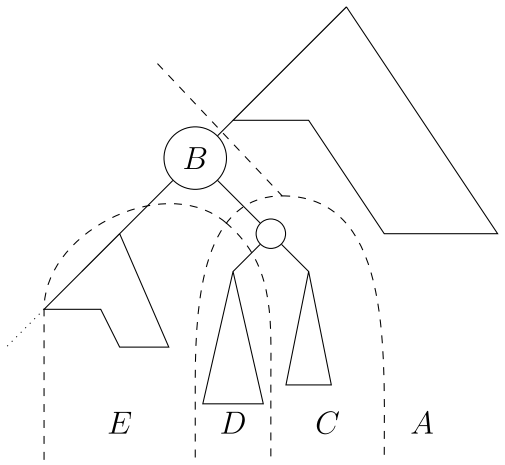


Figure 9.4: A graphical representation of case five for $\mathcal{M}$; see Lemma 81 and also Figure 9.3. Note that A, C, and E represent open terms and D a closed one. The term DCB then is open again.

5. $\mathcal{N}(l, l', r'-1) \cup \mathcal{N}(l'+1, r', r) \cup \mathcal{O}(l, m, r) \cup \mathcal{R}(l, m, r)$

Further the sets involved in the unions of case 4 and 5 are disjoint.

Proof. This proof is similar to the previous one, only case five slightly differs. Again, either the interval is completely contained in one of the three subintervals for which we fall back to $\mathcal{N}(l, l', r'-1)$, $\mathcal{N}(l'+1, r', r)$, or $\mathcal{N}(l'+1, m, r'-1)$ respectively.

Otherwise let $A = \mathcal{N}(l, l', r'-1) \cup \mathcal{N}(l'+1, r', r)$, similar to the previous proof. Notice that by Lemma 80 we obtain $\mathcal{N}(l, l', r'-1) \cup \mathcal{N}(l'+1, r', r)$ is a disjoint union and an interval. Following the line of the previous proof, if we are in the $AB\circledast$ situation, then case 4 holds as $\circledast$ is outside of $[l, r]$. In case of $BA\circledast$, B is not part of $\mathcal{N}$ due to the maximality of $\mathcal{N}^1$. If A is closed, we can insert $\circledast$ by $\mathcal{O}(l, m, r)$ and obtain an open term. Open terms following $\mathcal{O}(l, m, r)$ can be appended and are addressed by $\mathcal{R}(l, m, r)$.

This is possible since $\mathcal{M}^2 = \mathcal{N}^2$, which we know from Lemma 80. □

The intervals $\mathcal{M}$ and $\mathcal{N}$ are built around the property of containing a middle position m. The intervals $\mathcal{L}$ and $\mathcal{R}$ are different: They can lie arbitrarily within $[l, l'-1]$, or $[r'+1, r]$ respectively, and we initially do know nothing about the location of the middle positions. Our goal is to reduce $\mathcal{L}$ and $\mathcal{R}$ to some $\mathcal{M}(\bar{l}, \bar{m}, \bar{r})$ where we find $\bar{l}$, $\bar{m}$, and $\bar{r}$ using a binary search.

Lemma 83. *Given a term T and subinterval $[l, r]$ with middle m then for $\mathcal{L}$ there exists an interval $[\bar{l}, \bar{r}] \subseteq [l, l'-1]$ with middle $\bar{m}$ that can be found by binary search such that $\mathcal{L} = \mathcal{M}(\bar{l}, \bar{m}, \bar{r})$.*

Proof. By definition the interval $\mathcal{L}$ lies left to $\mathcal{M} \cup \mathcal{N}$. The set $\mathcal{M} \cup \mathcal{N}$ is a closed term and $\mathcal{M} \cup \mathcal{N} \cup \mathcal{O}$ is an open one by the definition of $\mathcal{L}$. We then want to address the largest term in $[l, l'-1]$ that comes before $\mathcal{M}$. We can use $\mathcal{M}(\bar{l}, \bar{m}, \bar{r})$ for this. The inclusion $\mathcal{L} \subseteq \mathcal{M}(\bar{l}, \bar{m}, \bar{r})$ is clear from the maximality of $\mathcal{M}(\bar{l}, \bar{m}, \bar{r})$. On the other hand, the converse direction is also true since any position to the right of $\mathcal{L}$ is rooted after l'.

Now, we can use the binary search inside $[l, l'-1]$: Start with this interval and then recursively do the following: If $\bar{m}$ is the middle position of the current interval $[\bar{l}, \bar{r}]$ then continue with:

- Search in the left half $[\bar{l}, \bar{m}-1]$ if $\mathcal{L}$ is entirely left of $\bar{m}$.
- Search in the right half $[\bar{m}+1, \bar{r}]$ if $\mathcal{L}$ is entirely right of $\bar{m}$.
- $\mathcal{L} = \mathcal{M}(\bar{l}, \bar{m}, \bar{r})$ if $\mathcal{L}$ contains $\bar{m}$, hence we may stop.

□

Lemma 84. *Given a term T and subinterval $[l, r]$ with middle m then for $\mathcal{R}$ there exists an interval $[\bar{l}, \bar{r}] \subseteq [r'+1, r]$ with middle $\bar{m}$ that can be found by binary search such that $\mathcal{R} = \mathcal{M}(\bar{l}, \bar{m}, \bar{r})$.*

Proof. This proof is similar to the previous one. First, note that $\mathcal{R} \subseteq \mathcal{M}(\bar{l}, \bar{m}, \bar{r})$ and $\mathcal{R}^2 = \mathcal{M}^2(\bar{l}, \bar{m}, \bar{r})$ because of maximality. Moreover, $\mathcal{R}$ is an open term. Now, if $\mathcal{M}(\bar{l}, \bar{m}, \bar{r})$ is a strict superset it must contain the operation set $\mathcal{O}(l, m, r)$. Inside $[r'+1, r]$ both descendants stay open, so there is no open term in $[r'+1, r]$ that contains $\mathcal{O}(l, m, r)$.

The binary search is the same as in the previous proof. □

9.3 The Evaluation Algorithm

The algorithm we present is based on a recursion leading to a parallel algorithm, which we present in form of a circuit construction. Lemmas 81, 82, 83, and 84 directly suggest how the recursive evaluation will work: To evaluate an interval we compute smaller fixed subintervals and then use the results to obtain the overall result. The functional algebra $\mathcal{F}(\mathcal{A})$ then allows us to combine the recursively obtained values.

In particular we need the following parts:

- Conversion of the term into PNF.

- Decision procedures determining for given intervals which case holds. By 'case' we mean the ones from Lemma 81 and 82.
- The actual evaluation using the PNF and the computed cases.

The first step is the PNF conversion, which can be found in [Bus87]. The conversion is of complexity $\mathbf{TC}^0$. The resulting term is T.

For the evaluation we need to implement circuits that on a given interval $[l, r]$ evaluate the intervals $\mathcal{M}$, $\mathcal{N}$, $\mathcal{L}$, and $\mathcal{R}$. In the case of $\mathcal{M}$, we need to distinguish whether an open or a closed term is evaluated. In the first case the output is a value of the domain $\mathbb{D}$ and in the second it is a function of $\widetilde{\mathbb{D}}$. In the other cases the result is always a function. These are the names of the evaluation circuits:

- $\textsc{Eval}_{\text{closed}}(\mathcal{M}(l, m, r))$
- $\textsc{Eval}(\mathcal{M}(l, m, r))$
- $\textsc{Eval}(\mathcal{N}(l, m, r))$
- $\textsc{Eval}(\mathcal{L}(l, m, r), \bar{l}, \bar{m}, \bar{r})$
- $\textsc{Eval}(\mathcal{R}(l, m, r), \bar{l}, \bar{m}, \bar{r})$

The variables $\bar{l}, \bar{m}, \bar{r}$ exist to serve the binary search as mentioned in Lemma 83 and 84.

These circuits all work in a similar way: Depending on the structure of the term one of a number of cases holds, which determines how the output value is composed from the recursion results. So, the recursion results are combined for each of the cases and these combination results are then fed into a multiplexer-gate, which chooses the right output.

The circuits determining the cases are:

- $\textsc{Case}(\mathcal{M}(l, m, r))$
- $\textsc{Case}(\mathcal{N}(l, m, r))$
- $\textsc{Case}(\mathcal{L}(l, m, r), \bar{l}, \bar{m}, \bar{r})$
- $\textsc{Case}(\mathcal{R}(l, m, r), \bar{l}, \bar{m}, \bar{r})$

In the end, $\textsc{Eval}_{\text{closed}}(\mathcal{M}(1, \lfloor n/2 \rfloor, n))$ is the circuit evaluating the whole term. For all recursive definitions of circuits, assume some look-up table construction if the interval becomes smaller than some constant. Also, if an open interval is evaluated and it happens to be empty, the identity function is returned as a result.

9.3.1 Evaluating the $\mathcal{M}$-Interval

This part is based on Lemma 81. Consider its five cases:

1. $\mathcal{M} = \mathcal{M}(l, l', r' - 1)$
2. $\mathcal{M} = \mathcal{M}(l' + 1, r', r)$
3. $\mathcal{M} = \mathcal{M}(l' + 1, m, r' - 1)$
4. $\mathcal{M} = \mathcal{M}(l, l', r' - 1) \cup \mathcal{N}(l' + 1, r', r)$
5. $\mathcal{M} = \mathcal{L}(l, m, r) \cup \mathcal{M}(l, l', r' - 1) \cup \mathcal{N}(l' + 1, r', r) \cup \mathcal{O}(l, m, r) \cup \mathcal{R}(l, m, r)$

The circuit $\textsc{Case}(\mathcal{M}(l, m, r))$ determines which case holds for given l, m and r. It actually has five output bits - one for each case. The circuit for the i-th output bit is $\textsc{Case}_i(\mathcal{M}(l, m, r))$. Instead of actually stating a circuit we specify MAJ[<] formulas for each output. This is sufficient since MAJ[<] is contained in $\mathbf{TC}^0$ which in turn is a subset of $\mathbf{NC}^1$. Also note that $\lhd_T$ is also expressible in MAJ[<] logic [Bus87].

$$\begin{aligned}\textsc{Case}_1(\mathcal{M}(l, m, r)) = {} & \exists x \;\; m \le x < r' \land (\exists y \;\; l \le y < l' \land y \lhd_T x) \\ & \land \forall x \;\; r' \le x \le r \Rightarrow \neg(\exists y \;\; l \le y < m \land y \lhd_T x)\end{aligned}$$

$$\begin{aligned}\textsc{Case}_2(\mathcal{M}(l, m, r)) = {} & \exists x \;\; r' \le x \le r \land (\exists y \;\; l' < y \le m \land y \lhd_T x) \\ & \land \forall x \;\; r' \le x \le r \Rightarrow \neg(\exists y \;\; l \le y \le l' \land y \lhd_T x)\end{aligned}$$

$$\begin{aligned}\textsc{Case}_3(\mathcal{M}(l, m, r)) = {} & \exists x \;\; m \le x < r' \land (\exists y \;\; l' < y \le m \land y \lhd_T x) \\ & \land \forall x \;\; r' \le x \le r \Rightarrow \neg(\exists y \;\; l \le y \le m \land y \lhd_T x) \\ & \land \forall x \;\; m \le x < r' \Rightarrow \neg(\exists y \;\; l \le y \le l' \land y \lhd_T x)\end{aligned}$$

$$\begin{aligned}\textsc{Case}_4(\mathcal{M}(l, m, r)) = {} & \exists x \;\; r' < x \le r \land \exists y \;\; l \le y < l' \land y \lhd_T x \\ & \land \forall u \forall v \;\; x \le u \land (x < u \lor v < y) \Rightarrow \neg(v \lhd_T u) \\ & \land \exists z \;\; l' \le z < r' \land y \lhd_T z \land z + 1 \lhd_T x\end{aligned}$$

$$\begin{aligned}\textsc{Case}_5(\mathcal{M}(l,m,r)) = {} & \exists x \;\; r' < x \leq r \wedge \exists y \;\; l \leq y < l' \wedge y \lhd_T x \\ & \wedge \forall u \forall v \;\; x \leq u \wedge (x < u \vee v < y) \Rightarrow \neg(v \lhd_T u) \\ & \wedge \exists z \exists u \exists v \;\; y \lhd_T v \wedge v + 1 \lhd_T z \\ & \wedge z + 1 \lhd_T u \wedge u + 2 \lhd_T x \\ & \wedge v + 1 \lhd_T u + 1\end{aligned}$$

Now that we have the means of deciding the case of Lemma 81 for a given interval, we can actually evaluate the interval. We receive the evaluation results recursively for the intervals $\mathcal{M}(l, l', r'-1)$, $\mathcal{M}(l'+1, r', r)$, $\mathcal{M}(l'+1, m, r'-1)$, $\mathcal{N}(l'+1, r', r)$, $\mathcal{L}(l, m, r)$, and $\mathcal{R}(l, m, r)$. By combining these we are able to obtain the output value.

- In cases one to three the combination is trivial as we only pass a recursively computed value.

- In case four, for the output of $\textsc{Eval}_{\text{closed}}(\mathcal{M}(l,m,r))$ we use a functional application gate ($\odot$) which gets the results from $\textsc{Eval}_{\text{closed}}(\mathcal{M}(l,l',r'-1))$ and $\textsc{Eval}(\mathcal{N}(l'+1,r',r))$. For the output of $\textsc{Eval}(\mathcal{M}(l,m,r))$ we use a composition gate ($\circ$) which gets the outputs of $\textsc{Eval}(\mathcal{M}(l,l',r'-1))$ and $\textsc{Eval}(\mathcal{N}(l'+1,r',r))$.

- Case five is composed as

  $$\mathcal{L}(l,m,r) \cup \mathcal{M}(l,l',r'-1) \cup \mathcal{N}(l'+1,r',r) \cup \mathcal{O}(l,m,r) \cup \mathcal{R}(l,m,r).$$

  The subinterval $\mathcal{M}(l,l',r'-1) \cup \mathcal{N}(l'+1,r',r)$ is a term and can be obtained just like in case four. For interval $\mathcal{M}(l,l',r'-1) \cup \mathcal{N}(l'+1,r',r) \cup \mathcal{O}(l,m,r)$ we use that result and feed it together with an identity function into a $\widetilde{\circledast}$-gate if $\mathcal{O}(l,m,r)$ points to a symbol $\circledast$. Then we take this value and the result of $\textsc{Eval}(\mathcal{R}(l,m,r))$ and feed it into a composition gate, which then yields the value for $\mathcal{M}(l,l',r'-1) \cup \mathcal{N}(l'+1,r',r) \cup \mathcal{O}(l,m,r) \cup \mathcal{R}(l,m,r)$. Finally, we take this value and compose it with the result of $\textsc{Eval}(\mathcal{L}(l,m,r))$ to get the value for the whole interval; see Figure 9.5.

In case five we need a multiplexer construction to select the right operation $\circledast$, i.e. we do the construction for all possible operations and then select the right one using the multiplexer, which is directed by the following:

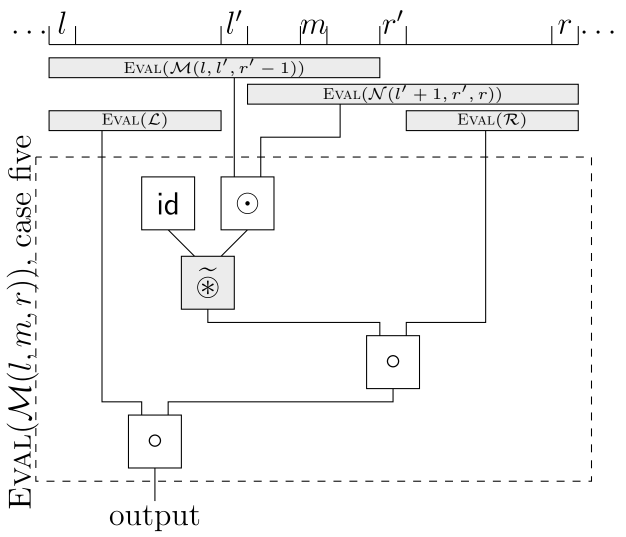

$\dots l$
l'
m
r'
$r \dots$
Eval($\mathcal{M}(l, l', r' - 1)$)
Eval($\mathcal{N}(l' + 1, r', r)$)
Eval($\mathcal{L}$)
Eval($\mathcal{R}$)
Eval($\mathcal{M}(l, m, r)$), case five
id
$\odot$
$\widetilde{\circledast}$
$\circ$
$\circ$
output

Figure 9.5: The dashed box represents the subcircuit of Eval($\mathcal{M}(l, m, r)$), which performs the combination in case five. Note that the box $\widetilde{\circledast}$ corresponds to the operation symbol $\circledast$ in position B in figures 9.3 and 9.4. This box actually is not a single gate but also a construction, which is shown in Figure 9.6.

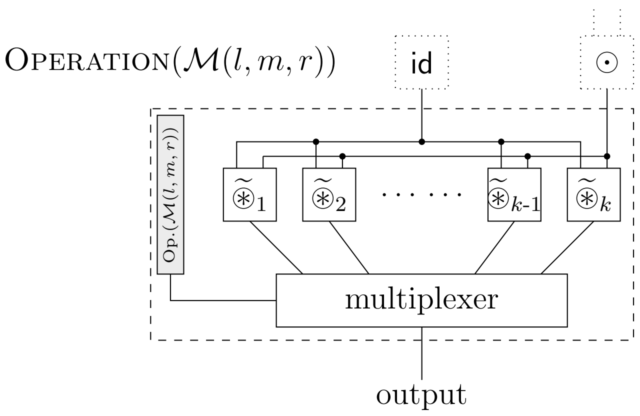
$$\begin{aligned}
\textsc{Operation}_{\circledast}(\mathcal{M}(l,m,r)) = {} & \exists x \ \ r' < x \le r \land \exists y \ \ l \le y < l' \land y \lhd_T x \\
& \land \forall u \forall v \ \ x \le u \land (x < u \lor v < y) \Rightarrow \neg(v \lhd_T u) \\
& \land \exists z \exists u \exists v \ \ y \lhd_T v \land v + 1 \lhd_T z \\
& \land z + 1 \lhd_T u \land u + 2 \lhd_T x \\
& \land v + 1 \lhd_T u + 1 \\
& \land Q_{\circledast}(u + 1)
\end{aligned}$$

This is the same as the formula for $\textsc{Case}_5(\mathcal{M}(l, m, r))$ but it also checks whether $\circledast$ is in the position contained in $\mathcal{O}(l, m, r)$; see Figure 9.6.

Finally, as we have these five possible combinations we use a multiplexer gate and the results of $\textsc{Case}_i(\mathcal{M}(l, m, r))$ to select the right one as output; see Figure 9.7.

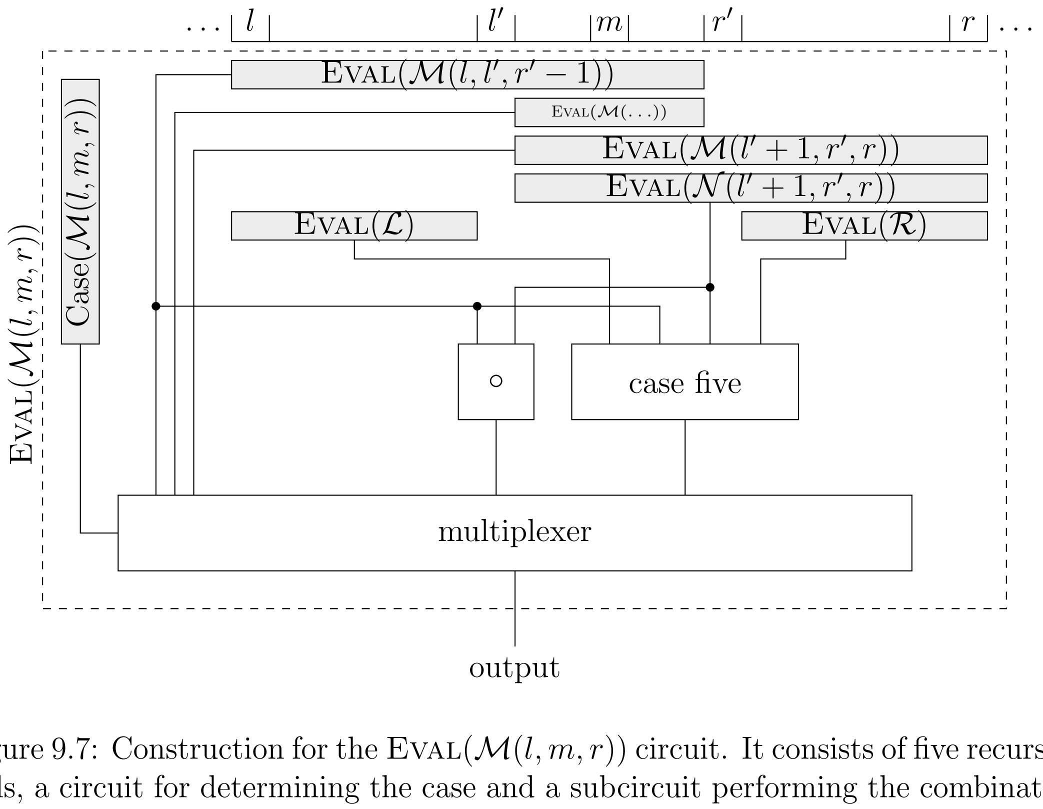
9.3.2 Evaluating the $\mathcal{N}$-Interval

The evaluation of the $\mathcal{N}$-intervals is very similar to the one previously described for $\mathcal{M}$. First, we only evaluate open terms in this case. One difference is that we need to have adjusted circuits $\textsc{Case}(\mathcal{N}(l, m, r))$ computing the case:



$\textsc{Operation}(\mathcal{M}(l,m,r))$
id
$\odot$
$\text{Op.}(\mathcal{M}(l,m,r))$
$\widetilde{\circledast}_1$
$\widetilde{\circledast}_2$
$\cdots\cdots$
$\widetilde{\circledast}_{k\text{-}1}$
$\widetilde{\circledast}_k$
multiplexer
output

Figure 9.6: In case five of the computation of $\textsc{Eval}(\mathcal{M}(l,m,r)$, the operation has to be computed and used. Figure 9.5 shows where the operation circuit shown here has to be inserted.



$\dots\ l\ \ l'\ \ m\ \ r'\ \ r\ \dots$
$\textsc{Eval}(\mathcal{M}(l,l',r'-1))$
$\textsc{Eval}(\mathcal{M}(\ldots))$
$\textsc{Eval}(\mathcal{M}(l'+1,r',r))$
$\textsc{Eval}(\mathcal{N}(l'+1,r',r))$
$\textsc{Eval}(\mathcal{L})$
$\textsc{Eval}(\mathcal{R})$
$\text{Case}(\mathcal{M}(l,m,r))$
$\textsc{Eval}(\mathcal{M}(l,m,r))$
$\circ$
case five
multiplexer
output

Figure 9.7: Construction for the $\textsc{Eval}(\mathcal{M}(l,m,r))$ circuit. It consists of five recursive calls, a circuit for determining the case and a subcircuit performing the combination for case five as shown in Figure 9.5.

$$\begin{aligned}
\textsc{Case}_1(\mathcal{N}(l,m,r)) = \exists y\ \ & l \le y \le l' \wedge l-1 <_T y-1 \wedge \neg(l-1 \le_T y) \\
\wedge\ \forall u\ \ & (l \le u \le m \wedge l-1 <_T u-1 \wedge \neg(l-1 \le_T u)) \Rightarrow u \le y \\
\wedge\ \exists x\ \ & m \le x < r' \wedge y \lhd_T x \\
\wedge\ \forall v\ \ & x < v \le r \Rightarrow \neg(y \lhd_T v)
\end{aligned}$$

$$\begin{aligned}
\textsc{Case}_2(\mathcal{N}(l,m,r)) = \exists y\ \ & l' < y \le m \wedge l-1 <_T y-1 \wedge \neg(l-1 \le_T y) \\
\wedge\ \forall u\ \ & (l \le u \le m \wedge l-1 <_T u-1 \wedge \neg(l-1 \le_T u)) \Rightarrow u \le y \\
\wedge\ \exists x\ \ & r' \le x < r \wedge y \lhd_T x \\
\wedge\ \forall v\ \ & x < v \le r \Rightarrow \neg(y \lhd_T v)
\end{aligned}$$

$$\begin{aligned}
\textsc{Case}_3(\mathcal{N}(l,m,r)) = \exists y\ \ & l' < y \le m \wedge l-1 <_T y-1 \wedge \neg(l-1 \le_T y) \\
\wedge\ \forall u\ \ & (l \le u \le m \wedge l-1 <_T u-1 \wedge \neg(l-1 \le_T u)) \Rightarrow u \le y \\
\wedge\ \exists x\ \ & m \le x < r' \wedge y \lhd_T x \\
\wedge\ \forall v\ \ & x < v \le r \Rightarrow \neg(y \lhd_T v)
\end{aligned}$$

$$\begin{aligned}
\textsc{Case}_4(\mathcal{N}(l,m,r)) = \exists y\ \ & l \le y \le l' \wedge l-1 <_T y-1 \wedge \neg(l-1 \le_T y) \\
\wedge\ \forall u\ \ & (l \le u \le m \wedge l-1 <_T u-1 \wedge \neg(l-1 \le_T u)) \Rightarrow u \le y \\
\wedge\ \exists x\ \ & r' \le x < r \wedge y \lhd_T x \\
\wedge\ \forall v\ \ & x < v \le r \Rightarrow \neg(y \lhd_T v) \\
\wedge\ \exists w\ \ & l' \le w < r' \wedge y \lhd_T w \wedge w+1 \lhd_T x
\end{aligned}$$

$$\begin{aligned}
\textsc{Case}_5(\mathcal{N}(l,m,r)) = \exists y\ \ & l \le y \le l' \wedge l-1 <_T y-1 \wedge \neg(l-1 \le_T y) \\
\wedge\ \forall u\ \ & (l \le u \le m \wedge l-1 <_T u-1 \wedge \neg(l-1 \le_T u)) \Rightarrow u \le y \\
\wedge\ \exists x\ \ & r' \le x < r \wedge y \lhd_T x \\
\wedge\ \forall v\ \ & x < v \le r \Rightarrow \neg(y \lhd_T v) \\
\wedge\ \exists w \exists z\ \ & l' \le w < r' \wedge y \lhd_T w \wedge w+1 \lhd_T z \wedge z+2 \lhd_T x
\end{aligned}$$

Now by applying Lemma 82, we can build $\textsc{Eval}(\mathcal{N}(l,m,r))$. Consider the cases:

1. $\mathcal{N} = \mathcal{N}(l, l', r' - 1)$
2. $\mathcal{N} = \mathcal{N}(l' + 1, r', r)$
3. $\mathcal{N} = \mathcal{N}(l' + 1, m, r' - 1)$
4. $\mathcal{N} = \mathcal{N}(l, l', r' - 1) \cup \mathcal{N}(l' + 1, r', r)$
5. $\mathcal{N} = \mathcal{N}(l, l', r' - 1) \cup \mathcal{N}(l' + 1, r', r) \cup \mathcal{O}(l, m, r) \cup \mathcal{R}(l, m, r)$

The construction for $\textsc{Eval}(\mathcal{N}(l, m, r))$ is similar to the one for $\textsc{Eval}(\mathcal{M}(l, m, r))$ with the exception that we use the appropriate recursive calls and do not use the $\mathcal{R}$-interval. Also, of course, we use $\textsc{Case}(\mathcal{N}(l, m, r))$ instead of $\textsc{Case}(\mathcal{M}(l, m, r))$.

9.3.3 Evaluating the $\mathcal{L}$-Interval

The key idea of evaluating an interval in our algorithm is that we evaluate the largest subterm in the interval that contains the middle. If we want to evaluate an $\mathcal{L}$-interval, we face the problem that it may lie arbitrarily within the considered interval such that is does not contain the middle. So, the idea is that we perform a binary search in order to find a interval whose middle is part of $\mathcal{L}$; see Lemma 83.

Our search interval will be $[\bar{l}, \bar{r}]$ with middle $\bar{m}$. We then distinguish three cases:

1. $\bar{m} \in \mathcal{L}$
2. $\mathcal{L} \subseteq [\bar{l}, \bar{m} - 1]$
3. $\mathcal{L} \subseteq [\bar{m} + 1, \bar{r}]$

In the first case we can fall back to $\textsc{Eval}(\mathcal{M}(\bar{l}, \bar{m}, \bar{r}))$. In the second we recurse using

$$\textsc{Eval}(\mathcal{L}(l, m, r), (\bar{l}, \bar{l} + \bar{m} - 1)/2, \bar{m} - 1)$$

and in the third case we use

$$\textsc{Eval}(\mathcal{L}(l, m, r), (\bar{m} + 1, (\bar{r} + \bar{m} + 1)/2, \bar{r}).$$

So, we have three recursive calls, which we feed into a multiplexer gate. The multiplexer gate is directed by $\textsc{Case}(\mathcal{L}(l, m, r), \bar{l}, \bar{m}, \bar{r})$, which decides which of the tree cases hold:

$$\begin{aligned}\textsc{Case}_1(\mathcal{L}(l,m,r),\bar{l},\bar{m},\bar{r}) = {} & \textsc{Case}_5(\mathcal{M}(l,m,r)) \wedge y \leq \bar{m} \leq v \\ = {} & \exists x \;\; r' < x \leq r \wedge \exists y \;\; l \leq y < l' \wedge y \lhd_T x \\ & \wedge \forall u \forall v \;\; x \leq u \wedge (x < u \vee v < y) \Rightarrow \neg(v \lhd_T u) \\ & \wedge \exists z \exists u \exists v \;\; y \lhd_T v \wedge v+1 \lhd_T z \\ & \wedge z+1 \lhd_T u \wedge u+2 \lhd_T x \\ & \wedge v+1 \lhd_T u+1 \\ & \wedge y \leq \bar{m} \leq v\end{aligned}$$

$$\textsc{Case}_2(\mathcal{L}(l,m,r),\bar{l},\bar{m},\bar{r}) = \textsc{Case}_5(\mathcal{M}(l,m,r)) \wedge y \leq v < \bar{m}$$

$$\textsc{Case}_3(\mathcal{L}(l,m,r),\bar{l},\bar{m},\bar{r}) = \textsc{Case}_5(\mathcal{M}(l,m,r)) \wedge \bar{m} < y \leq v$$

9.3.4 Evaluating the $\mathcal{R}$-Interval

Evaluating an $\mathcal{R}$-interval is again very similar to $\mathcal{L}$. For $\textsc{Eval}(\mathcal{R}(l,m,r),\bar{l},\bar{m},\bar{r})$ we use the same multiplexer construction for the binary search as in $\textsc{Eval}(\mathcal{L}(l,m,r),\bar{l},\bar{m},\bar{r})$ and only have to adjust the case computation:

$$\textsc{Case}_1(\mathcal{R}(l,m,r),\bar{l},\bar{m},\bar{r}) = \textsc{Case}_5(\mathcal{M}(l,m,r)) \wedge u+2 \leq \bar{m} \leq x$$

$$\textsc{Case}_2(\mathcal{R}(l,m,r),\bar{l},\bar{m},\bar{r}) = \textsc{Case}_5(\mathcal{M}(l,m,r)) \wedge u+2 \leq x \leq \bar{m}$$

$$\textsc{Case}_3(\mathcal{R}(l,m,r),\bar{l},\bar{m},\bar{r}) = \textsc{Case}_5(\mathcal{M}(l,m,r)) \wedge \bar{m} \leq u+2 \leq x$$

9.4 Complexity and Correctness: Proof of Theorem 75

The correctness of our construction follows from the lemmas of Section 9.2 since we only directly implemented those.

Our circuit construction uses the kind of gates, which we may use for $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$ circuits. We used multiplexer gates of three respectively five inputs instead of two, which can be implemented easily.

The construction also stays in logarithmic depth with regard to the input length: The PNF conversion is computable in the bounds of $\mathbf{TC}^0$. The same is true for the case computations. The evaluation circuits entail the case circuits as well as recursive calls. In every call the range becomes smaller by at least a factor of $2/3$, so the depth is logarithmic.

Analyzing the size of our construction, we see that we use a polynomial number of circuits, which originate in MAJ[<] formulas, which each result in polynomial size circuits. Each recursive evaluation circuit covers a certain subinterval and since there is only a quadratic number of subintervals, we get the polynomial bound for the whole construction.

Lastly, we indicate DLOGTIME-uniformity. As it is usually done, we have to show how to assign addresses to gates and then state $\mathsf{FO}[<,+,\times]$ formulas, which take such addresses and tell what function some gate is assigned as well as how the gates are wired. Consider a circuit $\textsc{Eval}(\mathcal{M}(l,m,r))$. It consists of several recursively defined subcircuits and a fixed number of extra gates to combine the results of the subcircuits, which we call the combination gates of $\textsc{Eval}(\mathcal{M}(l,m,r))$. An addressing scheme can look like this: We assign each $\textsc{Eval}(\mathcal{M}(l,m,r))$ circuit a word w and for the six subcircuits we assign words $w000$, $w001$, $w010$, $w011$, $w100$, and $w101$. We address the finitely many gates we use for combining the recursively obtained values, which are left by $w\$x$ where x is a unique word for each occurring gate. One can easily see that this scheme can be applied for all kinds of circuits we defined.

Now, it is easy to come up with a $\mathsf{FO}[<,+,\times]$ formula that assigns each gate its type. On an input $w\$x$ it only takes a look-up to which kind of gate x corresponds. The wiring between gates can also be expressed: For a pair of combination gates of some $\textsc{Eval}(\mathcal{M}(l,m,r))$, the task is again just a look-up in a table. If we have a pair such that one is the output of a recursion, we can also model this by looking at the last letter of w in the address $w\$x$. In the case of small intervals $r-l$, the computation $\textsc{Eval}(\mathcal{M}(l,m,r))$ becomes a look-up table, which accesses input gates, which we can also model. The output gate is a gate with an address of the form $\$x$ for an appropriate x.

Note that we also have circuits like $\textsc{Case}(\mathcal{M}(l,m,r))$, which are given in terms of MAJ[<] formulas. By [BL06] we know that these are also in DLOGTIME-uniform $\mathbf{NC}^1$.

9.5 Conclusion

Summary

This chapter was purely dedicated to the parallel evaluation algorithm. First, we defined the functional algebra $\mathcal{F}(\mathcal{A})$ based on an algebra $\mathcal{A}$, which captures precisely what is needed for parallel computation: There is a linear time algorithm for evaluation, which uses the operations of $\mathcal{A}$, but if one wants to parallelize, the cost of having the more complex algebra $\mathcal{F}(\mathcal{A})$ has to be paid. Therefore, our main result is an $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$ upper bound for evaluation. The proof is a direct circuit construction, which is defined recursively.

Contributions

The content of this chapter has been already published in [KLL17a, KLL17b]. In its generality it is entirely new, however, it draws from the algorithm of Buss [Bus87] and its subsequent version [BCGR92]. Our result extends to arbitrary algebras and its proof is somewhat different. While it is still locally quite technical, the overall proof structure is simple, especially compared to Buss's approach. An important tool to formulate and prove the main result is the functional algebra $\mathcal{F}(\mathcal{A})$, which captures precisely what is needed to make the tradeoff from linear time to logarithmic depth.

Sources and Related Work

Our algorithm for the term evaluation problem fits in the long chain of contributions dedicated to the term evaluation problem. The origins can be vaguely traced back to the investigation about upper bounds for the Boolean formula value problem. In [Lyn77], Lynch studied it first and achieved a log-space upper bound. Subsequently Cook conjectured that this bound is tight [Coo85] which, as we know today, it is not (unless log space equals log depth). Earlier, a way to deal with formulas that are very deep trees was investigated by Spira [Spi71]: By a quadratic increase in size one can balance a Boolean formula. Brent built upon this work [Bre74]. Going from balancing to obtaining an **NC** (in fact, $\mathbf{NC}^1$, i.e. log-depth) upper bound is not difficult. It is known that if the transformation can be done in $\mathbf{NC}^1$, the evaluation is in $\mathbf{NC}^1$.

Cook and Gupta [Gup85] as well as Ramachandran [Ram86] were the next in line who showed that $\mathcal{O}(\log n \log\log n)$ deep circuits suffice for evaluating formulas. Based on [Gup85], Buss showed an ALOGTIME bound [Bus87] which equals logarithmic depth [Ruz81] and is known to be tight. His proof utilized a sophisticated two-player pebbling game.

From there on the research proceeded in the direction of broadening the scope of the result. This continued research is always rooted in the work of [Gup85] and [Bus87]. Many other interesting works have contributed to this rich line of research, each solving the term evaluation problem over a specific algebra [Meh80], [Dym88], [KLM10], [KLL16].

It should be pointed out that the PNF normal form we defined originates also in [Bus87] where it is called PLOF. Also, we are aware of a simplified version [Bus93] of [Bus87] which directly operates on the infix notation instead of the normal form, however, we found the normal-form to be more convenient.

There already exists follow-up work based on the findings of this chapter [GL17] which gives an alternate proof, but is more geared towards term balancing. This, however, is merely a different perspective on the same matter. The paper also makes use of parts of the machinery we presented, most notably the functional algebra $\mathcal{F}(\mathcal{A})$.

Further Research

First, one could do a more detailed analysis of our construction. Then, improvements could be done in terms of simplifying the construction or reducing the complexity, for example, by lowering the degree of the polynomial of the size bound. One could also try to obtain lower bounds.

Chapter 10

Applications of Evaluation

In the previous chapter we saw that a term can be evaluated over some algebra $\mathcal{A}$ in $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$. This is a rather unusual complexity class that suits to the evaluation problem, yet it is not immediately clear how $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$ relates to classical complexity classes. This is a problem we tackle in this chapter. Moreover, the evaluation problem is just one example amongst others, however, it is a very useful one since many problems that have a parallel algorithm can be reduced to an evaluation problem. This already outlines the plan for upcoming sections: We offer a framework that yields upper bounds for certain problems in terms of parallel computation. The steps, which we will go into more detail in a moment, are roughly the following: First, reduce the problem in question to an evaluation problem over an algebra $\mathcal{A}$. Second, embed $\mathcal{F}(\mathcal{A})$ in an appropriate algebra $\mathcal{B}$ such that we can perform the third step, which is showing the actual upper bound for $\mathcal{B}$-$\mathbf{NC}^1$, which is done by analyzing the complexity of the operations.

To make this formally precise we have to pay attention to some circumstances. First of all, the input is always Boolean, i.e. a word over some alphabet Σ. In general, a problem is a function $\Sigma^* \to \mathbb{D}$. The word problem, for example, embeds via $\mathbb{D} = \mathbb{B}$. We reduce the problem to term evaluation over an algebra $\mathcal{A} = (\mathbb{D}; O)$ and hence the $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$-circuit we get through Theorem 75, really outputs a $\mathbb{D}$-value and not one that is necessarily a natural number or Boolean.

To ease the following constructions, note that it does not add complexity if we allow the input to be evaluated over some family of algebras instead of only one algebra. If we modify Theorem 75, we get:

Theorem 85. *Given an algebra $\mathcal{A}$, which may be a family of possibly many-sorted algebras $\mathcal{A} = (\mathcal{A}_n)_{n\in\mathbb{N}}$ with $\mathcal{A}_n = (\mathbb{D}(n); O(n))$, where all $\mathcal{A}_n$ have the same signature*

σ, then the evaluation function $\mathrm{eval}_{\mathcal{A}}\colon \mathbb{T}(\sigma) \to \bigcup_{n\in\mathbb{N}} \mathbb{D}(n)$ *is in DLOGTIME-uniform* $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$.

We evaluate over a family of algebras, which means that an input of length n is evaluated over $\mathcal{A}_n$. The domain of $\mathcal{A}_n$ is $\mathbb{D}(n)$, which may consist of several subdomains. The input terms are all of the same signature hence we impose the same signature on the algebras $\mathcal{A}_n$. The evaluation function $\mathrm{eval}_{\mathcal{A}}$ naturally extends to families of algebras, and so does $\mathcal{F}(\mathcal{A})$. If $\mathcal{A} = (\mathcal{A}_n)_{n\in\mathbb{N}}$ is a family of algebras, $\mathcal{F}(\mathcal{A})$ is the family $(\mathcal{F}(\mathcal{A}_n))_{n\in\mathbb{N}}$.

Now suppose we showed that some problem is in $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$. For the embedding step of $\mathcal{F}(\mathcal{A})$ we use the notion of divisibility. We have to find some family of algebras $\mathcal{B} = (\mathcal{B}_n)_{n\in\mathbb{N}}$ such that $\mathcal{F}(\mathcal{A}_n) \preceq \mathcal{B}_n$. By definition, there exists an epimorphism $\phi'_n\colon \mathcal{B}_n \to \mathcal{F}(\mathcal{A}_n)$. Now, by Theorem 23 there exists an isomorphism ϕ_n between $\mathcal{F}(\mathcal{A}_n)$ and $\mathcal{B}_n/\ker(\phi'_n)$.

If we have found a family of algebras $\mathcal{B}$ as described, we may reduce the problem to $\mathcal{B}/\ker(\phi)$-$\mathbf{NC}^1$, but usually, we choose $\mathcal{B}$ in a way such that showing an upper bound is easy. Also, we do not care if $\mathcal{B}$ is a little bit too big. So, we actually go with $\mathcal{B}$-$\mathbf{NC}^1$. If the evaluation of the input term t is $d \in \mathbb{D}$, then the $\mathcal{B}$-$\mathbf{NC}^1$ circuit will output a representative of the set ${\phi'}_{|t|}^{-1}(d)$. This output also may be represented differently than d. For example, while d is some natural number, the output of the $\mathcal{B}$-$\mathbf{NC}^1$ circuit could be the binary representation of d with some leading zeros. Hence, in this example ${\phi'}_{|t|}^{-1}(d)$ might be the set of binary representations of d with an arbitrary number of leading zeros.

The previous example motivates introducing the notion of *coding* since this is what actually happens: We have some algebra and code the values e.g. in binary. For $n = |t|$, the inverse homomorphism ${\phi'}_n^{-1}\colon \mathbb{D}_n \to 2^{\mathcal{B}_n}$ is close to what we want to consider a coding: An algebra value is assigned a set of valid representations. When doing the actual constructions, it usually does not matter which of the many representations we choose, so we let a coding c_n be a map $\mathcal{F}(\mathcal{A}_n) \to \mathcal{B}_n$ such that $c_n(d) \in \phi'_n(d)$. The map c_n is a monomorphism. A coding c for the whole problem then is a family $c = (c_n)_{n\in\mathbb{N}}$. Later, rather than defining $\mathcal{B}$ and then c, we will only define the coding of the operations or the coding of the domains since usually one already defines the other. If, for example, we have a coding for the operations, then we may choose $\mathcal{B}$ as just the image

$$c(\mathcal{A}) = (c(\mathbb{D}_1), \ldots, c(\mathbb{D}_S); \circledast_1^c, \ldots, \circledast_k^c).$$

Note that c being a generalized homomorphism assigns each operation in $\mathcal{A}$ a term over $\mathcal{B}$. The functions these terms represent become the operations of $c(\mathcal{A})$. Note that $c(\mathcal{A})$ might be smaller than $\mathcal{B}$ which, however, is no problem in the applications since $c(\mathcal{A})$-$\mathbf{NC}^1 \subseteq \mathcal{B}$-$\mathbf{NC}^1$.

Now, we have the $c(\mathcal{F}(\mathcal{A}))$-$\mathbf{NC}^1$-circuit, which is evaluating the terms. These circuits usually are Boolean or arithmetic ones. The algebra $c(\mathcal{F}(\mathcal{A}))$ has the subdomains $c(\mathbb{D})$ and $c(\widetilde{\mathbb{D}})$. If c was chosen well, we may proceed with showing the upper bound. The $c(\mathcal{F}(\mathcal{A}))$-$\mathbf{NC}^1$-circuit internally makes computations involving the operations of $c(\mathcal{F}(\mathcal{A}))$. Suppose we want to show an upper bound in terms of $\mathbf{NC}^i$, the goal is now to simulate the operations by constructions using Boolean wires and gates of bounded fan-in. In general, if the operations are in $\mathbf{NC}^i$ then the overall problem is in $\mathbf{NC}^{i+1}$. If the operations are in $\mathbf{AC}^i$, then the overall problem is in $\mathbf{AC}^{i+1}$. If the operations are in $\#\mathbf{NC}^i_{\mathbb{N}}$, then the overall problem is in $\#\mathbf{NC}^{i+1}$. Along the same lines we obtain the cases for $\mathbf{SAC}^i$ and Gap-classes.

In summary, the previous considerations led to the following template for proving upper bounds. Note that we allow for unary operations here even though Theorem 75 and the definition of $\mathcal{F}(\mathcal{A})$ only allow for either 0-ary or binary ones. Unary operations, however, may be simulated through binary ones. So, if $\circledast\colon \mathbb{D} \to \mathbb{D}$, we let $\widetilde{\circledast}\colon \widetilde{\mathbb{D}} \to \widetilde{\mathbb{D}}$ be defined in the obvious way.

1. **Find an algebra $\mathcal{A}$** and reduce the problem $P\colon \Sigma^* \to \mathbb{D}$ to a term evaluation problem over $\mathcal{A}$; for convenience $\mathcal{A}$ might be a family. That means, for example, if we want a many-one reduction we have to find a map $f\colon \Sigma^* \to \mathbb{T}(\sigma(\mathcal{A}))$ such that $P = \mathrm{eval}_{\mathcal{A}} \circ f$. Other reduction types may also be used, of course.

2. **Find a coding c for $\mathcal{F}(\mathcal{A})$** such that we can find meaningful complexity upper bounds for the operations in $c(\mathcal{F}(\mathcal{A}))$. In our applications this means that $c(\mathcal{F}(\mathcal{A}))$ has domains that are vectors, matrices and cross-products over either $\mathbb{B}$, $\mathbb{N}$, or $\mathbb{Z}$. These can be represented just as sequences of bits or numbers. This leads to Boolean or arithmetic circuits.

3. **Analyze the complexity of the operations used in $c(\mathcal{F}(\mathcal{A}))$-$\mathbf{NC}^1$.** We then get the according complexity for P provided that the complexity of the reduction step does not dominate the overall complexity. As a summary, recall that for $\mathcal{A} = (\mathbb{D}; B, Z)$ we have $\mathcal{F}(\mathcal{A}) = (\mathbb{D}, \widetilde{\mathbb{D}}; B, Z, \widetilde{B}, \circ, \odot, \mathrm{id})$, so we have to analyze the operations of

$$c(\mathcal{F}(\mathcal{A})) = (c(\mathbb{D}), c(\widetilde{\mathbb{D}}); B^c, Z^c, \widetilde{B}^c, \circ^c, \odot^c, \mathrm{id}^c) :$$

   - The non-constant operations of $\mathcal{A}$: For each $\circledast \in B$ we have to analyze

$$\circledast^c\colon c(\mathbb{D}) \times c(\mathbb{D}) \to c(\mathbb{D}).$$

   - The functional versions of the non-constant operations of $\mathcal{A}$. So, for each $\circledast \in B$ we have to analyze

$$\widetilde{\circledast}^c\colon c(\widetilde{\mathbb{D}}) \times c(\mathbb{D}) \to c(\mathbb{D}).$$

This also contains the unary operations, which can be simulated by binary ones.

- To spare us the hassle of expressing unary operations through binary ones way we include unary operations explicitly in this list:

$$\circledast^c\colon c(\mathbb{D}) \to c(\mathbb{D})$$

as a member of B^c and

$$\widetilde{\circledast}^c\colon c(\widetilde{\mathbb{D}}) \to c(\widetilde{\mathbb{D}})$$

as a member of $\widetilde{B}^c$.

- The functional composition of $\mathcal{F}(\mathcal{A})$:

$$\circ^c\colon c(\widetilde{\mathbb{D}}) \times c(\widetilde{\mathbb{D}}) \to c(\widetilde{\mathbb{D}}).$$

- The substitution operation of $\mathcal{F}(\mathcal{A})$:

$$\odot^c\colon c(\widetilde{\mathbb{D}}) \times c(\mathbb{D}) \to c(\mathbb{D}).$$

The algebra also has 0-ary operations, but for these there is no complexity to analyze.

Multiplexer operations are not part of the algebra, but come into play in the construction of the $c(\mathcal{F}(\mathcal{A}))$-$\mathbf{NC}^1$ circuits. Usually, their complexity analysis is trivial:

- Multiplexer operations for all subdomains X of $c(\mathcal{F}(\mathcal{A}))$:

$$\mathrm{mp}_X\colon \mathbb{B} \times X \times X \to X$$

These operations are used in the $c(\mathcal{F}(\mathcal{A}))$-$\mathbf{NC}^1$ circuit as black boxes. In this third step we have to come up with an efficient implementation of all these operations in order to derive a good overall upper bound. The depth increases by a logarithmic factor when comparing the complexity of the functions and the overall circuit. So, if, say, all the functions are in $\#\mathbf{NC}^1_{\mathbb{D}}$, then $c(\mathcal{F}(\mathcal{A})) \subseteq \#\mathbf{NC}^2$.

There is also another interpretation of the template, in particular of the second and third step. The first step provides us with a $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$ bound. In the resulting circuits there are wires and gates that process algebra values. Now, the second step of the template is replacing the wires and the third step is replacing the algebra gates by actual constructions.

If an operation $\circledast_i$ is not commutative, we actually have also a symmetric variant present as enforced by the PNF definition. In the complexity analysis we can omit those operations because we can use the same algorithm for those variants by just switching the two inputs.

In the upcoming applications we will strictly follow the template. Without explicitly mentioning it, all upper bounds we show are meant in terms of DLOGTIME-uniform circuits.

10.1 The Boolean Formula Value Problem and Finite Algebras

The *Boolean formula value problem* (BFVP) is the problem of evaluating Boolean formulas. That is evaluating terms over the algebra $\mathcal{B} = (\mathbb{B}; \wedge\vee, \neg, \bot, \top)$.

Theorem 86 ([Bus87])**.** *The Boolean formula value problem is in* $\mathbf{NC}^1$.

Proof. **1. step.** We do not need a reduction, since the problem directly is a evaluation problem over the algebra $\mathcal{B}$

2. step. Consider the algebra $\mathcal{F}(\mathcal{B}) = (\mathbb{B}, \widetilde{\mathbb{B}}; \wedge, \vee, \neg, \bot, \widetilde{\wedge}, \widetilde{\vee}, \widetilde{\neg}, \circ, \odot, \mathrm{id})$. Here, $\widetilde{\mathbb{B}} = \mathbb{B}^{\mathbb{B}}$ has four elements. We choose some coding c with $c(\mathbb{B}) = \mathbb{B}$ and $c(\mathbb{B}^{\mathbb{B}}) = \mathbb{B}^2$. The coding for the operations follows.

3. step. Consider the algebra $c(\mathcal{F}(\mathcal{B}))$. The operations of the subalgebra $c(\mathcal{B}) = \mathcal{B}$ can be implemented directly by single gates. The other operations need constant size circuits, i.e. $\mathbf{NC}^0$. The same is true for multiplexer gates. Hence $c(\mathcal{F}(\mathcal{B}))$-$\mathbf{NC}^1 \subseteq \mathbf{NC}^1$. □

In the previous proof we used that the algebra is finite. If it is finite, we only need constant size circuits to implement the operations. Thus, we may state a general theorem, which is folklore:

Theorem 87. *If $\mathcal{A}$ is a finite algebra, then evaluating terms over $\mathcal{A}$ is in* $\mathbf{NC}^1$.

10.2 Evaluating Arithmetic Terms and Distributive Algebras

We consider evaluating terms over $\mathcal{N} = (\mathbb{N}; +, \times, 0, 1)$ and $\mathcal{Z} = (\mathbb{Z}; +, \times, 0, 1)$.

Theorem 88 ([BCGR92])**.** *Evaluating terms over $\mathcal{N}$ is in* $\#\mathbf{NC}^1$.

Proof. **1. step.** The problem is directly a term evaluation problem, hence no reduction is needed and we stick to the algebra $\mathcal{N}$.

2. step. Consider the algebra

$$\mathcal{F}(\mathcal{N}) = (\mathbb{N}, \widetilde{\mathbb{N}}; +, \times, 0, 1, \widetilde{+}, \widetilde{\times}, \circ, \odot, \mathrm{id}).$$

Here, we have $\widetilde{\mathbb{N}} \subseteq \mathbb{N}^{\mathbb{N}}$. The functions in $\widetilde{\mathbb{N}}$ are of the form $x \mapsto ax + b$ for some $a, b \in \mathbb{N}$. We choose a coding c such that $c(\mathbb{N}) = \mathbb{N}$ and $c(\widetilde{\mathbb{N}}) = \mathbb{N}^2$ and begin with the identity function $x \mapsto 1x + 0$, which is clearly of this form. It has to be shown that the operations of $\mathcal{F}(\mathbb{N})$ leave functions in this form.

- $\circ^c$: Given some functions $f(x) = a_f x + b_f$ and $g(x) = a_g x + b_g$, then $f \circ g$ is also of the desired form: $x \mapsto a_f a_g x + a_f b_g + b_f$. So, $c(f \circ g) = c(f) \circ^c c(g) = (a_f, b_f) \circ^c (a_g, b_g) = (a_f a_g, a_f b_g + b_f)$.
- $\widetilde{+}^c$: Consider $c(f \widetilde{+} e)$ for $f \in \mathbb{N}^{\mathbb{N}}$ and $e \in \mathbb{N}$. Now, $c(f) \widetilde{+}^c c(e) = (a, b) \widetilde{+}^c e = (a, b + e)$ where $f(x) = ax + b$.
- $\widetilde{\times}^c$: Consider $c(f \widetilde{\times} e)$ for $f \in \mathbb{N}^{\mathbb{N}}$ and $e \in \mathbb{N}$. Now, $c(f) \widetilde{\times}^c c(e) = (a, b) \widetilde{\times}^c e = (a \times e, b \times e)$ where $f(x) = ax + b$.

This shows that c is indeed a valid coding.

3. step. We now have an upper bound of $c(\mathcal{F}(\mathbb{N}))$-$\mathbf{NC}^1$. As all operations use constantly many inputs of natural numbers, there exist arithmetic circuit implementations for all operations. Further, all Boolean gates and multiplexer gates can be simulated by arithmetic circuit constructions, so all operations are in $\#\mathbf{NC}^0_{\mathbb{N}}$. Hence, we get $c(\mathcal{F}(\mathbb{N}))$-$\mathbf{NC}^1 \subseteq \#\mathbf{NC}^1$. $\square$

The same construction carries over to integers:

Theorem 89. *Evaluating terms over $\mathcal{Z}$ is in Gap*$\mathbf{NC}^1$.

In the previous proof we used distributivity of $+$ and $\times$, which allows us to represent functions by two values. We can generalize this idea. We call an algebra $\mathcal{A} = (\mathbb{D}; \circledast_1, \dots, \circledast_k)$ *distributive* if $i < j$ implies

$$(d_1 \circledast_j d_2) \circledast_i d_3 = (d_1 \circledast_i d_3) \circledast_j (d_2 \circledast_i d_3),$$

where we assume without loss of generality that the operations are in an order fulfilling the equation. If that is the case, we find a representation of $\widetilde{\mathbb{D}}$ as $c(\widetilde{\mathbb{D}}) = \mathbb{D}^k$ because we can choose the maps as $x \mapsto (\dots((x \circledast_1 d_1) \circledast_2 d_2) \dots \circledast_k d_k$. No matter what operations we perform now, we can rearrange the resulting term again in this form. So, computing those functions is not harder than the original algebra, which gives us a modified version of Theorem 75:

Theorem 90. *Evaluating terms over a distributive algebra $\mathcal{A}$ is in $\mathcal{A}$-$\mathbf{NC}^1$.*

10.3 Automata

In the first part of this work we introduced a host of different automaton models. Now, it is time to analyze their complexity. For ordinary automata this means analyzing the complexity of the word problem. For other kinds it means analyzing the complexity of the functions the automaton implements.

We will not show the complexity for every single automaton model explicitly. For example, as we saw before, ranked tree automata do nothing else than evaluate a term over a finite algebra. Also, there are the nested word automata: We leave them out in favor of visibly pushdown automata, since they are so closely related. The problems for unranked tree automata can be reduced to the equivalent problems for VPAs.

10.3.1 Language Recognizing Automata

We show complexity bounds for word problems of automata. There are actually two versions of the word problem we may consider. Classically, we fix an automaton $\mathcal{M}$ as ask for the complexity of determining whether $\mathcal{M}$ accepts an input. Additionally, there is also the *uniform word problem* in which also $\mathcal{M}$ is part of the input. Naturally, the complexity of the uniform word problem is as least as high as the complexity of the word problem.

We will generalize the word problem by looking at the counting problem:

Theorem 91. *Given a non-deterministic VPA $\mathcal{M}$ and a well-matched word $w \in \mathrm{WM}(\hat{\Sigma})$ as inputs, then computing the number of accepting runs of $\mathcal{M}$ on w is in* $\#\mathbf{SAC}^1$

Proof. **1.step.** We assume that the automaton has a state set $[n]$ where n is the input length. Choosing so is no restriction. A well-matched word can be considered to be a linearization of a tree or a term. So, what we will do is to interpret the input word as a term over a family of algebras $(\mathcal{A}_n)_{n\in\mathbb{N}}$ with

$$\mathcal{A}_n = (\mathbb{N}^{[n]\times[n]}; \circledast, (\otimes_{a,b})_{a\in\Sigma_{\mathrm{call}}, b\in\Sigma_{\mathrm{ret}}}, (\dagger_e)_{e\in\Sigma_{\mathrm{int}}\cup\{\epsilon\}}),$$

whose operations will be defined in a moment. Given a well-matched input word w, we construct a term $t(w)$. If the input w is either the empty word or a internal letter, then the corresponding term is $t(w) = \dagger_w$. If $w = w_1w_2$ where w_1, w_2 are well matched, then $t(w) = t(w_1) \circledast t(w_2)$. If $w = aw'b$ where $a \in \Sigma_{\text{call}}$, $b \in \Sigma_{\text{ret}}$, and w' is well-matched, then $t(w) = \otimes_{a,b} t(w)$. The elements of the domain $\mathbb{N}^{[n]\times[n]}$ then assign each pair of states q_1, q_2 the number of runs from q_1 to q_2 there are by passing through the corresponding well-matched word. The definition of the algebra operations in particular is as follows:

- $\dagger_\epsilon$ is a 0-ary operation, hence an element of the domain, which is a function $[n] \times [n] \to \mathbb{N}$. We define it as $(q, q') \mapsto 1$ if and only if $q = q'$ and $(q, q') \mapsto 0$ otherwise.
- $\dagger_e$ for $e \in \Sigma_{\text{int}}$ is defined as $(q, q') \mapsto 1$ if $q' \in \delta_{\text{int}}(q, e)$ and $(q, q') \mapsto 0$ otherwise.
- $\circledast$ is a binary operation and $\alpha \circledast \beta$ is defined as $(\alpha \circledast \beta)(q, q') = \sum_{r\in[n]} \alpha(x, r)\beta(r, y)$.
- $\otimes_{a,b}$ is unary and $(\otimes_{a,b}\alpha)(q, q')$ is defined as the sum of all $\alpha(p, p')$ for which there exists $\gamma \in \Gamma$ such that $(p, \gamma) \in \delta_{\text{call}}(q, a)$ and $q' \in \delta_{\text{ret}}(p', b, \gamma)$.

If we evaluate the term over this algebra, we get the number of runs of $\mathcal{M}$ on w.

2. step. The algebra $\mathcal{F}(\mathcal{A}_n)$ has a subdomain, which consists of maps of the form $\mathbb{N}^{[n]\times[n]} \to \mathbb{N}^{[n]\times[n]}$. Potentially, the set of such maps is too large, but actually they are made up in a regular manner. The idea for a function $[n] \times [n] \to \mathbb{N}$ was to store how many paths there are between a pair of states for a given well-matched word. For functions $\mathbb{N}^{[n]\times[n]} \to \mathbb{N}^{[n]\times[n]}$ there is a similar picture: Given a well-matched word uv where u and v do not necessarily have to be well-matched, i.e. (u, v) is a context, then a function $f \in \widetilde{\mathbb{D}}$ is storing how many ways there are for given states q_1, q_2, q_3, q_4 from q_1 to q_2 via u and from q_3 to q_4 via v. If we consider $f(d)$ where d is a function $d\colon [n] \times [n] \to \mathbb{N}$, then d fills in the transitions from q_2 to q_3. If d resulted from evaluating a well matched word w, then $f(d)$ is the evaluation corresponding to w_1ww_2.

The idea for the following coding c is that we we have to store natural numbers for these four-tuples of states. We set

$$c\left(\mathbb{N}^{[n]\times[n]}\right) = \mathbb{N}^{n\times n}$$

and

$$c\left(\left(\mathbb{N}^{[n]\times[n]}\right)^{\mathbb{N}^{[n]\times[n]}}\right) = (\mathbb{N}^{n\times n})^{n\times n}.$$

To assign a semantic to these matrices we define $\odot^c$ first:

- $\odot^c$: Given $c(f) \in (\mathbb{N}^{n\times n})^{n\times n}$ and $c(d) \in \mathbb{N}^{n\times n}$ we define the matrix $c(f(d)) = c(f \odot d) = c(f) \odot^c c(d) = A$. For a matrix like $A \in \mathbb{N}^{n\times n}$ we write $A(q_1, q_2)$ to address the entry, which corresponds to the pair q_1, q_2. If we are given a matrix like $c(f)$, we write $c(f)(q_1, q_2)$ to address the matrix corresponding to q_1, q_2 and we set $c(f)(q_1, q_2)(q_3, q_4) = c(f)(q_1, q_2, q_3, q_4)$. Now, $A(q_1, q_2)$ is defined as

$$\sum_{q_3, q_4 \in [n]} (c(f)(q_1, q_2, q_3, q_4)) \, (c(d)(q_3, q_4)) \, .$$

This is the sum of the entries of the point-wise matrix multiplication of $c(f)(q_1, q_2)$ and $c(d)$. Note that the coding of the identity map is $c(\mathrm{id}) = I^{n \times n}$, where I is the identity map of size n times n.

- $\circ^c$: Given $c(f)$ and $c(g)$ of $(\mathbb{N}^{n \times n})^{n \times n}$, then

  $$(c(f) \circ^c c(g)) \, (q_1, q_2, q_3, q_4) = \sum_{q_5, q_6 \in [n]} (c(f)(q_1, q_2, q_5, q_6)) \, (c(g)(q_5, q_6, q_3, q_4)) \, .$$

- $\circledast^c$: This is just the normal matrix multiplication.

- $\otimes^c_{a,b}$: Let the matrix $M_{a,b} \in (\mathbb{N}^{n \times n})^{n \times n}$ be defined such that $M_{a,b}(q_1, q_2, q_3, q_4) = 1$ if there exists $\gamma \in \Gamma$ with $(q_2, \gamma) \in \delta_{\mathrm{call}}(q_1, a)$ and $q_4 \in \delta_{\mathrm{ret}}(q_3, b, \gamma)$ and otherwise $M_{a,b}(q_1, q_2, q_3, q_4) = 0$. Now, we set $\otimes^c_{a,b} c(d) = M_{a,b} \odot^c c(d)$.

- $\widetilde{\otimes}^c_{a,b}$: This is similar to the previous case and we set $\widetilde{\otimes}^c_{a,b} c(f) = M_{a,b} \circ^c c(f)$.

- $\widetilde{\circledast}^c$: We set $c(f \widetilde{\circledast} d) = c(f) \widetilde{\circledast}^c c(d)$ as

  $$c(f \widetilde{\circledast} d)(q_1, q_2) = \sum_{q_3 \in [n]} c(f)(q_1, q_3) c(d)(q_3, q_2)$$

  where the summation is a point-wise matrix summation and the multiplication is a scalar multiplication.

3. step. Up to now, we have reduced the problem in a way that we know it is in $c(\mathcal{F}(\mathcal{A}))$-$\mathbf{NC}^1$. By considering the definition of the algebra operations above, one can see that in all cases arithmetic circuits of constant depth suffice. In particular we only use multiplication between two elements. The fan-in of addition gates is n, which is the number of states of the automaton. Hence, we have a $\#\mathbf{SAC}^0_{\mathbb{N}}$ bound for the operations. This in turn yields the bound of $\#\mathbf{SAC}^1$ for the actual problem. $\square$

If we look at the previous proof, it is easy to see how transitioning from the counting to the Boolean case effects the complexity. The following result for the uniform word problem can be derived:

Theorem 92. *The uniform word problem for non-deterministic VPAs is in* $\mathbf{SAC}^1$.

On the other hand, if we stay in the counting case but fix the automaton, we get the following:

Theorem 93 ([KLM12])**.** *For a fixed non-deterministic VPA, counting the number of accepting runs is in* $\#\mathbf{NC}^1$.

Finally, the classical word problem is obtained by considering the Boolean case and fixing the automaton:

Theorem 94 ([Dym88])**.** *For a fixed VPA, the word problem is in* $\mathbf{NC}^1$.

The proof for the last theorem is immediate. To solve the word problem, we simply have to evaluate the input within the syntactic forest algebra. This algebra is finite, and so by Theorem 87 we get the $\mathbf{NC}^1$ bound.

10.3.2 Weighted Automata

Next, we show the complexity of wighted automata, in particular for weighted VPAs (WVPAs). The result is independent of the underlying semiring.

Theorem 95. *Functions implemented by WVPAs over a semiring* $R = (\mathbb{D}; \oplus, \otimes)$ *are in* R-$\mathbf{NC}^1$.

Proof. **1. step.** In a WVPA, for all computations the sum of weights is obtained by $\oplus$ and then these sums are then multiplied using $\otimes$. An approach of doing the computation in that order is awkward since there can exist exponentially many runs. However, since we have a semiring at hand we can use distributivity. Again, we interpret the input word as a term over an appropriate algebra. Then we can assign each well-matched factor w a value, which is a map $Q \times Q \to \mathbb{D}$ where $(q_1, q_2) \mapsto d$ represents what the weight is that is accumulated when going from q_1 to q_2 by reading w. This idea is related to the to one for the construction for counting paths in VPAs. Similarly, as the algebra we choose:

$$\mathcal{A} = \left(\mathbb{D}^{Q\times Q}; \circledast, (\circledcirc_{a,b})_{a\in\Sigma_{\text{call}}, b\in\Sigma_{\text{ret}}}, (\dagger_e)_{e\in\Sigma_{\text{int}}\cup\{\epsilon\}}\right)$$

where

$$(f \circledast f)(q_1, q_2) = \bigotimes_{q\in Q} (f(q_1, q) \oplus g(q, q_2)) .$$

The operation $\dagger_e$ is 0-ary. For $\dagger_\epsilon$ we define $\dagger_\epsilon(q, q')$ to be 0 if $q = q'$ and 1 otherwise. Further,

$$\circledcirc_{a,b}(f)(q_1, q_2) = \bigotimes_{q_1', q_2' \in Q, \gamma\in\Gamma} (\mathsf{weight}(q_1, q_1', a, \gamma) \oplus f(q_1', q_2') \oplus \mathsf{weight}(q_2', q_2, b, \gamma)) .$$

Here, $\mathsf{weight}\colon Q \times Q \times \Sigma_{\text{call}} \cup \Sigma_{\text{ret}} \times \Gamma \to \mathbb{D}$ assigns each transition its weight; if $a \in \Sigma_{\text{call}}$ then $\gamma \in \Gamma$ is the letter that is pushed onto stack and if $b \in \Sigma_{\text{ret}}$ it is the one popped off stack.

A given well-matched input word w, results into the term $t(w)$. If $w = \epsilon$, then $t(w) = \dagger_\epsilon$ and if $w = c$ for $c \in \Sigma_{\text{int}}$, then $t(w) = \dagger_c$. For $w = w_1 w_2$, where w_1 and w_2 are also well-matched we have $t(w) = t(w_1) \circledast t(w_2)$. If $w = aw'b$ for $a \in \Sigma_{\text{call}}$ and $b \in \Sigma_{\text{ret}}$, then $t(w) = \circledcirc_{a,b}(t(w'))$.

By induction one can see that the constructed term evaluates to the function that represents the weight for each pair of states. By assuming that there exists one initial and one final state, looking up at the pair of initial and final state we get the final output.

2. and 3. step. The algebra $\mathcal{F}(\mathcal{A})$ has a domain $\mathbb{D}^{Q\times Q}$ and one that consist of functions $\mathbb{D}^{Q\times Q} \to \mathbb{D}^{Q\times Q}$. Similar to the proof of Theorem 91 and because R is distributive, for $n = |Q|$ the coded domains $\mathbb{D}^{n\times n}$ and $(\mathbb{D}^{n\times n})^{n\times n}$ may be chosen. These domains again equate to constant-sized lists of $\mathbb{D}$-values. So, updating them requires circuits of constant size using R-gates. The result is a R-$\mathbf{NC}^1$ circuit family. □

Based on the previous proof, one can obtain the result for the case that the automaton is part of the input. Whereas $n = |Q|$ has been fixed, we now assume n to be the length of the input. We see that we need either $\otimes$-gates of unbounded fan-in or a logarithmically deep construction of binary $\otimes$-gates. If we do the latter, we get the following:

Theorem 96. *Given a WVPA $\mathcal{M}$ and a word $w \in \mathrm{WM}(\hat{\Sigma})$ as input over a fixed semiring $R = (\mathbb{D}; \oplus, \otimes)$, the problem of computing the output value of $\mathcal{M}$ on w is in R-$\mathbf{NC}^2$.*

Applied, we directly obtain:

Theorem 97. *Functions implemented by WVPA over $(\mathbb{N}; +, \times)$ are in $\#\mathbf{NC}^1$ and those over $(\mathbb{Z}, +, \times)$ are in Gap$\mathbf{NC}^1$.*

A prime example for $\mathcal{A}$ in the context of weighted automata is $(\mathbb{Z}; +, \min)$, hence:

Theorem 98. *Functions implemented by WVPA over $(\mathbb{Z}; +, \min)$ where the output is coded binary are in $\mathbf{SAC}^1$.*

Proof. By Theorem 95 we know that this problem is in $(\mathbb{Z}; +, \min)$-$\mathbf{NC}^1$. The class $\mathbf{SAC}^1$ is an upper bound because addition and minimum can be computed in Boolean circuits of constant-depth. □

10.3.3 Cost Register Automata

General cost register automata are not always very accessible for complexity analysis, so we will examine the restricted cases of copyless and polynomially bounded automata first. After that we consider one example of a CVPA model that is not restricted in that way. This model then has an algebra that makes it accessible again.

The following theorem relates to a result of [AM15]. In this paper it is shown that CCRA over free monoids have functions that can be computed in $\mathbf{NC}^1$. On the one hand we generalize this result from CRA to CVPA as well es from copylessness to polynomial boundedness. On the other hand, the bound is not as good. However, later we show that at least for CCVPA we get the same bound.

Theorem 99. *Functions implemented by polynomially bounded CVPAs over the free monoid* $(\Gamma^*; \circ)$ *are in* $\mathbf{TC}^1$.

Proof. **1. step.** Let $w \in \mathrm{WM}(\hat{\Sigma})$ be the input word. Like in Theorem 94, we will interpret the word as a term. First, we annotate the states of the run of the automaton $\mathcal{M}$ on w. Let $r \in (\Sigma \times Q)^{|w|}$ be such that $r_i = (w_i, q)$ where q is the state the automaton is in after $w_1 \dots w_{i-1}$ is read. Note that computing r is possible in $\mathbf{NC}^1$. Let X be the register set. We will show how $F'_{\mathcal{A}}(\mathcal{M})(w)\colon (\Gamma^*)^X \to (\Gamma^*)^X$ is computed from r. The image $F'_{\mathcal{A}}(\mathcal{M})(w)$, similarly as defined in Section 7.3.2, captures the register update function associated to the well matched word w. Now, $F_{\mathcal{A}}(\mathcal{M})(w) = \mu(q)(F'_{\mathcal{A}}(\mathcal{M})(w)(\nu_0))$ where q is the state the automaton is in after w is read and ν_0 is the initial valuation. For the upper bound we need to show the complexity of computing $F'_{\mathcal{A}}(\mathcal{M})(w)$.

To compute $F'_{\mathcal{A}}(\mathcal{M})(w)$, like in Theorem 94, we consider the well-matched word r to be a term t over the algebra $(\mathcal{A}_n)_{n\in\mathbb{N}}$ where

$$\mathcal{A}_n = \left(\left((\Gamma^*)^X\right)^{(\Gamma^*)^X}, \circledast, (\otimes_{(a,q_a),(b,q_b)})_{a\in\Sigma_{\text{call}}, b\in\Sigma_{\text{ret}}, q_a,q_b\in Q}, (\dagger_{e,q})_{q\in Q, e\in\Sigma_{\text{int}}\cup\{\epsilon\}}\right).$$

The domain $\mathbb{D} = ((\Gamma^*)^X)^{(\Gamma^*)^X}$ consists of functions that map valuations to valuations. The operation $\circledast$ is the concatenation of such functions. The operation $\otimes_{(a,q_a),(b,q_b)}$ is associated to the case $(a, q_a)r'(b, q_b)$ for $r' \in (\Sigma \times Q)^*$ being well-matched. Let γ be the symbol that is pushed onto the stack if a is read while being in state q_a. Then γ will be the symbol that is on the top of the stack when q_b is reached. Then $\otimes_{(a,q_a),(b,q_b)}$ is based on $\rho(q_b, b, \gamma)$.

The operation $\dagger_{\epsilon,q}$ is the identity function for all $q \in Q$. For $c \in \Sigma_{\text{call}}$, $\dagger_{c,q} \in \mathbb{D}$ is the register update function $\rho_{\text{int}}(q, a)$.

2. step. The algebra $\mathcal{F}(\mathcal{A})$ has the domains $\mathbb{D}$ and

$$\widetilde{\mathbb{D}} \subseteq \left(((\Gamma^*)^X)^{(\Gamma^*)^X}\right)^{((\Gamma^*)^X)^{(\Gamma^*)^X}} .$$

We will show that the elements of $\mathbb{D}$ are formed in such a way that they can be represented as words that contain placeholders for the registers. Notice that the size of the words in the registers is bounded by some polynomial p. So, for n being the input length, we choose $c(\mathbb{D}) = ((\Gamma \cup X)^{p(n)})^X$; we simultaneously consider it as a $|X|$-dimensional vector of words. For $\widetilde{\mathbb{D}}$ we observe the following: A context (u, v) evaluates to a function of $\widetilde{\mathbb{D}}$ that takes a function of $\mathbb{D}$, which in turn corresponds to the evaluation of a well-matched word w. The evaluation of uwv is then obtained by inserting the value and this result is a function of $\mathbb{D}$ corresponding. Now, this can be also considered differently. Let ν be the valuation that is present directly before uwv. The word u induces register updates that depend on ν, hence, if we think in terms of the coded domain, u induces words $(\Gamma \cup x)^*$ for all registers. To w corresponds also such words and we can combine them and get words for uw. In v we have access to the register valuations present after uw as well as intermediate results from within u; since those again depend on ν we may replace them directly. Hence, for every register, u induces a function that can be represented by a word $(\Gamma \cup X \cup X')^*$, where X' is a copy of X. The function corresponding to uwv can be obtained by beginning with the one for v and replacing the X' variables accordingly by the result of uw. So, for $c(\widetilde{\mathbb{D}})$ we choose $((\Gamma \cup X)^{p(n)})^X \times ((\Gamma \cup X \cup X')^{p(n)})^X$. By checking all operations of $\mathcal{F}(\mathcal{A})$, we see that this is actually a valid coding.

- $\circledast^c$: The coded operation $\circledast^c$ takes two vectors of words $c(d_1)$ and $c(d_2)$ and replaces every occurrence $x \in X$ in a word of $c(d_2)$ by $c(d_1)(x)$.

- $\otimes^c_{a,q_a,b,q_b}$: Let γ be the symbol that is pushed in state q_a if a is read. Now, $c(\otimes_{a,q_a,b,q_b,\gamma} d)$ is a determined by the following substitutions. First, $x \in X$ is substituted in all words of $c(d)$ by $\rho_{\text{call}}(q_a, a)(x)$; let the result be $e \in \mathbb{D}$. Then, in $\rho_{\text{ret}}(q_b, b, \gamma)$, every $x \in X$ is substituted by $e(x)$ and every $x_{\text{match}} \in X_{\text{match}}$ is substituted by x. The result of this last substitution is $c(f(d))$.

- $\dagger^c_{e,q}$: The coded version $\dagger^c_{e,q}$ is equal to $\rho(q, e)$ and $\dagger^c_\epsilon$ is the identity: $\dagger^c_\epsilon(x)$ is the word consisting of the single letter x.

- $\odot^c$: We already indicated how this operation works. If we are given $d \in \mathbb{D}$ and $f \in \widetilde{\mathbb{D}}$, then $f \odot d = f(d)$. Now, $c(f(d))$ is obtained as follows: Substitute the letters $x \in X$ in $c(d)$ by $c(f)_1(x)$; let the result be e and then substitute variables $x' \in X'$ by $e(x)$.

- $\circ^c$: Given $f, g \in \widetilde{\mathbb{D}}$ we define $c(f \circ g) = c(f) \circ^c c(g)$ as the pair where the first component is $c(g)_1$ in which every letter $x \in X$ is substituted by $c(f)_1(x)$.

The second component is obtained by substituting every letter $x \in X$ in $c(g)_2$ by $c(f)_1(x)$; let the result be e. Then by substituting every letter $x' \in X'$ in $c(f)_2$ by $e(x)$ we get the second component.

- $\widetilde{\circledast}^c$: Given a function $f \in \widetilde{\mathbb{D}}$ and some $d \in \mathbb{D}$, we have $c(f \widetilde{\circledast} d) = c(f) \widetilde{\circledast}^c c(d)$, which is again a pair in $c(\widetilde{\mathbb{D}})$. The first component is identical to $c(f)_1$. The second is obtained by replacing every $x \in X$ in $c(d)$ by $c(f)_2(x)$.
- $\widetilde{\otimes}^c_{a,q_a,b,q_b}$: Given $f \in \widetilde{\mathbb{D}}$, we have $c(\otimes_{a,q_a,b,q_b} f) = \otimes^c_{a,q_a,b,q_b} c(f)$. We can associate to a, q_a, b, q_b, γ a function g of $\widetilde{\mathbb{D}}$ such that $\widetilde{\otimes}_{a,q_a,b,q_b}(f) = g \circ f$.

3. step. All operations of the algebra $c(\mathcal{F}(\mathcal{A}))$ are based on substitutions in words. This problem is equivalent to computing the image of a free monoid homomorphism. It has been analyzed in [LM98] and hence we get a bound for the operations in $c(\mathcal{F}(\mathcal{A}))$, which is $\mathbf{TC}^0$. Because of polynomial boundedness the construction keeps polynomial size. This leads to the overall complexity of $\mathbf{TC}^1$.

□

The previous theorem can be used to obtain the following:

Theorem 100. *Polynomially bounded functions implemented by CVPAs over an algebra $\mathcal{A}$ are in $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$ or $\mathbf{TC}^1$, depending on which is the larger class.*

Proof. As the first step, we compute the image of the function over the term algebra $\mathcal{T}(\sigma(\mathcal{A}))$ instead of $\mathcal{A}$. In particular we do this by representing the terms as words. The previous theorem provides us with a $\mathbf{TC}^1$ bound for this step. After, the resulting term is evaluated using Theorem 75 which gives us the overall bound.

□

An application then is the case of the algebra $(\mathbb{Z}, \times, +)$.

Corollary 101. *Polynomially bounded functions implemented by CVPAs over the algebra $(\mathbb{Z}, \times, +)$ are in $\mathbf{TC}^1$.*

The previous findings are significant, however, their scope is limited to polynomial boundedness. If terms become too large, this approach does not work any more. If, on the other hand, the used algebra ensures that the values can be represented efficiently, we can go beyond the previous result. For example, a term $1 + 1 + \ldots + 1$ can be represented using linearly many bits, even when it is exponentially long. The next theorem is an example that exploits this observation.

Theorem 102 ([KLL16])**.** *Functions realized by CVPAs over $(\mathbb{Z}, +)$ are in Gap*$\mathbf{NC}^1$.

Proof. **1. step.** The first step is identical to the first step of Theorem 99 with the only difference that we consider $\mathbb{Z}$ instead of Γ^*. So, in this case we have

$$\mathcal{A}_n = \left(\left(\mathbb{Z}^X\right)^{\mathbb{Z}^X}, \circledast, (\otimes_{(a,q_a),(b,q_b)})_{a\in\Sigma_{\text{call}},b\in\Sigma_{\text{ret}},q_a,q_b\in Q}, (\dagger_{e,q})_{q\in Q,e\in\Sigma_{\text{int}}\cup\{\epsilon\}} \right).$$

2. step. The algebra $\mathcal{F}(\mathcal{A})$ has the domains $\mathbb{D}$ and

$$\widetilde{\mathbb{D}} \subseteq \left((\mathbb{Z}^X)^{\mathbb{Z}^X} \right)^{(\mathbb{Z}^X)^{\mathbb{Z}^X}}.$$

We will show that the elements of $\mathbb{D}$ are formed in such a way that they can be represented as m-dimensional matrices of integers, where $m = |X|$. Hence, the other domain $\widetilde{\mathbb{D}}$ consists of matrix-manipulating functions. These functions are of the form $x \mapsto AxB + C$ where A and B are matrices. So, we choose $c(\mathbb{D}) = \mathbb{Z}^{m\times m}$ and $c(\widetilde{\mathbb{D}}) = \mathbb{Z}^{m\times m} \times \mathbb{Z}^{m\times m} \times \mathbb{Z}^{m\times m}$. By checking all operations of $\mathcal{F}(\mathcal{A})$, we show that this is actually a coding.

- $\odot^c$: Given $d \in \mathbb{D}$ and $f \in \widetilde{\mathbb{D}}$, then $f \odot d = f(d)$. Now, $c(f)$ is a map $x \mapsto AxB+C$ and $c(d)$ is a matrix. Therefore, $c(f) \odot^c c(d) = c(f(d)) = Ac(d)B+C$.
- $\circ^c$: Given $f, g \in \widetilde{\mathbb{D}}$, then f is of the form $x \mapsto A_f x B_f + C_f$ and g is of the form $x \mapsto A_g x B_g + C_g$. Now, $c(f \circ g) = c(f) \circ^c c(g)$ is the map $x \mapsto A_f(A_g x B_g + C_g)B_f + C_f = A_f A_g x B_g B_f + A_f C_g B_f + C_f$, so $c(f \circ g) = (A_f A_g, B_g B_f, A_f C_g B_f + C_f)$.
- $\circledast^c$: The coded operation $\circledast^c$ takes two matrices and multiplies them.
- $\otimes^c_{a,q_a,b,q_b}$: This operation translates also into matrix multiplication. Let γ be the symbol that is pushed in state q_a if a is read. As by definition we have that $\otimes_{a,q_a,b,q_b,\gamma} f$ translates to $\rho_{\text{call}}(q_a, a) \circledcirc f \circledcirc \rho^1_{\text{ret}}(q_b, b, \gamma) + \rho^2_{\text{ret}}(q_b, b, \gamma)$. So, we define a matrix $M_{q_a,a}$ from $\rho(q_a, a)$ and matrices $M^1_{q_b,b,\gamma}$ and $M^2_{q_b,b,\gamma}$ from $\rho^1_{\text{ret}}(q_b, b, \gamma)$ and $\rho^2_{\text{ret}}(q_b, b, \gamma)$. Now, for $d \in \mathbb{D}$, we have $c(\otimes_{a,q_a,b,q_b,\gamma} d) = \otimes^c_{a,q_a,b,q_b,\gamma} c(d) = M_{q_a,a} c(d) M^1_{q_b,b,\gamma} + M^2_{q_b,b,\gamma}$.
- $\dagger^c_{e,q}$: This constant operation is a matrix $M_{e,q}$ corresponding to $\rho(q, e)$ and $\dagger^c_\epsilon$ is the identity matrix.
- $\widetilde{\circledast}^c$: Given a function $f \in \widetilde{\mathbb{D}}$ and some $d \in \mathbb{D}$, we have $c(f \widetilde{\circledast} d) = c(f) \widetilde{\circledast}^c c(d)$ where $\widetilde{\circledast}^c$ is again a multiplication: If $c(f)$ is given as $x \mapsto AxB + C$ then $c(f) \widetilde{\circledast}^c c(d)$ is $x \mapsto (AxB + C)c(D) = AxBc(D) + Cc(D)$, which is of the desired form.
- $\widetilde{\otimes}^c_{a,q_a,b,q_b}$: Given $f \in \widetilde{\mathbb{D}}$, we have $c(\otimes_{a,q_a,b,q_b} f) = \otimes^c_{a,q_a,b,q_b} c(f)$. If $c(f)$ is given as $x \mapsto AxB + C$, then $\otimes^c_{a,q_a,b,q_b,} c(f)$ is $x \mapsto M_{q_a,a}(AxB + C)M^1_{q_b,b,\gamma} + M^2_{q_b,b,\gamma} = M_{q_a,a} AxB M^1_{q_b,b,\gamma} + M_{q_a,a} C M^1_{q_b,b,\gamma} + M^2_{q_b,b,\gamma}$.

3. step. All operations of the algebra $c(\mathcal{F}(\mathcal{A}))$ are based on matrix operations and the domains are based on matrices of fixed dimensions. Because of that and since the matrices are of integer values, all operations of $c(\mathcal{F}(\mathcal{A}))$ are in Gap$\mathbf{NC}^0_{\mathbb{Z}}$. This leads to the upper bound of Gap$\mathbf{NC}^1$ for the problem in question. □

Similar to the previous theorem, one can prove that the functions of CVPAs over $(\mathbb{N}, +)$ are in $\#\mathbf{NC}^1$.

In [AM15] the complexity of copyless CRAs over $(\Gamma^*, \circ)$ has been determined to be $\mathbf{NC}^1$. The problem of Theorem 99 generalized this by considering CVPAs instead of CRAs and by relaxing copylessness to polynomial boundedness. Unfortunately the complexity then rises to $\mathbf{TC}^1$ in our proof. What we still can do, however, is to only focus on one generalization and determine the complexity of CCVPAs.

Theorem 103. *Functions implemented by CCVPAs over the free monoid* $(\Gamma^*; \circ)$ *are in* $\mathbf{NC}^1$.

Proof. By [AM15] we know that functions of CCRA over the free monoid are in $\mathbf{NC}^1$. This result can be used to obtain the result for CCVPA: Based on a CCVPA $\mathcal{M}$, we define a CCRA $\mathcal{M}'$. We also define a transduction $\tau\colon \Sigma^* \to \Sigma'^*$ such that $F_{(\Gamma^*;\circ)}(\mathcal{M})(w) = F_{(\Gamma^*;\circ)}(\mathcal{M}')(\tau(w))$ for all $w \in \Sigma^*$. Since τ will be computable in $\mathbf{NC}^1$, the result follows.

While reading a return letter a CCVPA may access register values form the matching position, but recall that an equivalent view on the matter is to think of this as a register value as being pushed onto the stack and later being popped off the stack when the matching return letter is read. An important insight for the construction is the fact that at any time during the computation, at most a constant number of register values can be stored on the stack that end up as a part of the final output; this constant is the number of registers. If there are more, then all but a constant number of then could also be replaced by a constant expression, which could be stored using the usual stack alphabet.

The main idea of τ is to make precomputations such that what is left, can be done by a CCRA. First of all, each position having a return letter should be labeled the stack symbol that is on the top of the stack when it is read. Secondly, we have to resolve the storing of register values on the stack. For that we use the observation form earlier and add registers for which instead of using the stack, the register values get stored in these additional registers and maintained such that they can be accessed in the matching return position. The transduction τ can label the word in a way that $\mathcal{M}'$ knows, when reading a return letter, where to find the appropriate register value.

Now, τ can be computed in $\mathbf{NC}^1$ and computing the output of the CCRA is also in $\mathbf{NC}^1$ which yields the overall complexity.

□

A simple consequence of the previous theorem is that computing (generalized) VPL-homomorphisms, i.e. forest algebra homomorphisms applied on VPLs, is in $\mathbf{NC}^1$. Such a homomorphism ϕ is determined by assigning every pair $(a, b) \in \Sigma_{\text{call}} \times \Sigma_{\text{ret}}$ a context (u, v) and assigning every $c \in \Sigma_{\text{int}}$ a well-matched word. Then on input w, $\phi(w)$ is computable in $\mathbf{NC}^1$. This is true because there exists a CCRA over the free monoid to solve that problem.

Similarly to Theorem 100 we get the following:

Theorem 104. *Functions implemented by CCVPAs over an algebra $\mathcal{A}$ are in $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$.*

Corollary 105. *Functions implemented by CCVPAs over the algebra $(\mathbb{Z}, \times, +)$ are in $\#\mathbf{NC}^1$.*

10.4 Circuits of Bounded Tree-Width

We apply the term evaluation algorithm to reprove a recent result about circuits of bounded tree-width [JS14]. It states that Boolean circuit families of polynomial size can be balanced to obtain logarithmically deep circuit families. We show a short and generalized proof using term evaluation.

Whenever we speak of tree-decompositions and tree-width of a circuit we mean it in correspondence to the graph of the circuit. The graph of a circuit satisfies some desirable properties, e.g. it is a DAG, which has input and output gates. As a tool we need to decompose the graphs of circuits in a way to preserve these properties which leads to the following lemma.

Lemma 106. *Given a graph G of a circuit C and a decomposition (T, τ) of G then there exists a decomposition (T', τ') of C with $\mathsf{width}(T', \tau') \in \mathcal{O}(\mathsf{width}(T, \tau))$ that satisfies:*

- *The tree T' is binary.*
- *If $u \in V(G)$ is a parent of v then let $p, q \in V(T')$ be the bags closest to the root satisfying $u \in \tau'(p)$ and $v \in \tau'(q)$. Then p is not closer to the root than q.*
- *For each input node $v \in V(G)$ there exists a leaf $l \in V(T')$ such that $v \in \tau'^{-1}(l)$.*
- *The output node of the circuit can be found in $\tau'^{-1}(r)$, where r is the root of the tree.*

Proof. We can assume the tree T' to be binary without increasing the width, because for minimal decompositions the maximal rank of nodes is dependent on the width, hence bounded and nodes with a rank greater than 2 can be resolved by a constant size construction.

The second requirement can be achieved by labeling the nodes by u that are labeled v and are closer to the root than all nodes labeled u.

The third requirement can be met by picking a node u labeled v and label the shortest path from u to some leaf with v. The last requirement can be implemented by labeling a path from a node labeled r to the root.

All modifications at most lead to a constant factor in the width. □

Through the lemma we see that assuming the stated properties preserves the boundedness of the tree-width. The proof idea for the following theorem is to interpret the tree-decomposition as a term and evaluate it over a fitting algebra.

Consider a circuit C_n of n inputs over an algebra $\mathcal{A} = (\mathbb{D}; O)$ and let $G = (V, E)$ be the graph of C_n. Let the tree-decomposition of minimal width be following the previous lemma and assume the width to be $w - 1$. We define the algebra

$$\mathcal{A}(C_n, w) = \left(\mathbb{D}'; (\circledast_{A,B,C})_{A,B,C \in \binom{V}{w}}, (\dagger_s)_{s \in S^{2w}}\right)$$

where $\mathbb{D}' = (\mathbb{D} \cup \{\bot\})^{2w}$, $\circledast_{A,B,C}$ is an operation $\mathbb{D}' \times \mathbb{D}' \to \mathbb{D}'$, and $\dagger_S$ is a constant operation where S consists of all values of 0-ary operations of $\mathcal{A}$ and $\bot$. Also let $A = \{a_{g_1}, \dots, a_{g_w}\}$, $B = \{b_{h_1}, \dots, b_{h_w}\}$ and $C = \{c_{i_1}, \dots, c_{i_w}\}$. We assume $V = [|V|]$ and $a_{g_j} < a_{g_{j+1}}$ for $j \in [|V| - 1]$. Similarly, we assume $b_{h_j} < b_{h_{j+1}}$ and $c_{i_j} < c_{i_{j+1}}$. Consider $\alpha \circledast_{A,B,C} \beta = \gamma$ where $\alpha, \beta, \gamma \in \mathbb{D}'$. For a node $a_{g_j} \in A$ the elements α_j and α_{w+j} correspond to the left and right parent of a_{g_j}. The situation for B and β, and respectively C and γ, is similar. The following rules define the operation $\circledast_{A,B,C}$ by specifying the result γ:

- For $c_{i_l} = a_{g_j}$, if $\gamma_l \neq \bot$, then $\gamma_l = \alpha_j$ and also if $\gamma_{w+l} \neq \bot$ then $\gamma_{w+l} = \alpha_{w+j}$
- For $c_{i_l} = b_{h_j}$, if $\gamma_l \neq \bot$, then $\gamma_l = \beta_j$ and also if $\gamma_{w+l} \neq \bot$ then $\gamma_{w+l} = \beta_{w+j}$.
- If c_{i_l} has the parents a_{g_j} and b_{h_k} then $\gamma_l = \alpha_j \circledast_a \alpha_{w+j}$ and $\gamma_{w+l} = \beta_k \circledast_b \beta_{w+k}$, for $\circledast_a$ being the operation of the gate a_{g_j} and $\circledast_b$ being the operation of the gate b_{h_k}.
- If a position in γ is not yet defined by the previous cases, we set it to $\bot$.

We only store inputs of gates. The overall output of a circuit, however, is not the input of any gate but only the output of a certain gate. To make the actual output value appear, we add a dummy gate, which receives the output value. From now on we assume this construction to be present.

As the sets A, B and C are finite and there are only finitely many possibilities of ways how the gates can be wired the consequence is that there is only a finite number of operations, which is independent of the actual circuit. Hence, we write $\mathcal{A}(w)$ while dropping the circuit in the notation.

Describing how to interpret the tree as a term is left. We begin with the decomposition of width $w-1$ satisfying the conditions of Lemma 106 and interpret it as a term over the algebra $\mathcal{A}(w)$ where each node v is assigned the operation $\circledast_{A,B,C}$ where $C = \tau(v)$ and B and C are the bags of the parents of v. So, every node in the tree becomes a binary operation in the term. This is also true for the leaves in the tree. To the left and to the right of the operations that correspond to a leaf there must be constants present. Such a constant s is a vector that is $\perp$ in all positions but those corresponding to an input gate; here the right input value is present. This then ends up being the operation $\dagger_s$.

The previous construction shows us that we can regard the tree-decomposition tree as a term that is equivalent to the original circuit:

Lemma 107. *Given a circuit C_n that has a tree-decomposition of with $w-1$ that satisfies the conditions of Lemma 106 and an input $x \in \mathbb{B}^n$, then $\mathrm{eval}_{\mathcal{A}}(C_n, x) = \pi_i(\mathrm{eval}_{\mathcal{A}(w)}(t))$ where t is the term we get from the tree-decomposition as described above and i is the position corresponding to the output gate in the result vector.*

We now immediately get the following:

Theorem 108. *For every family of circuits C of bounded tree-width w and polynomial size over an algebra $\mathcal{A}$ there exists an equivalent $\mathcal{F}(\mathcal{A}(w))$-$\mathbf{NC}^1$ circuit family.*

Proof. First, we use the previous lemma to get a term out of the circuit. For each input length there is one fixed term. We use the input to prepare the leaves of the term accordingly. Then evaluating those terms is in $\mathcal{F}(\mathcal{A}(w))$-$\mathbf{NC}^1$. Finally, the appropriate value of the result vector is the output. $\square$

The way we proved the previous theorem can be considered wasteful. For each input length there is only one term to be evaluated since it originates in one circuit for each input length, but we have the full evaluation machinery present which is unnecessary: All the subcircuits described in the previous chapter that serve deciding how to split the terms throughout the recursion could actually be precomputed and replaced by these precomputed results. That way we have effectively performed a balancing.

Thus, we may formulate the result also in the following way: Given a circuit C_n of n inputs and tree-width $w-1$ over an algebra $\mathcal{A}$, there exists an equivalent circuit T_n over $\mathcal{F}(\mathcal{A}(w))$ that is a balanced tree, i.e. $\mathrm{eval}_{\mathcal{A}}(C_n, x) = \pi_i(\mathrm{eval}_{\mathcal{A}}(T_n, x'))$ where i is the position corresponding to the output gate in the result vector and x' is the input adapted to be fed into T_n.

The construction can be applied to Boolean circuits:

Theorem 109 ([JS14])**.** *Languages accepted by families of Boolean circuits of polynomial size and bounded tree-width are in* $\mathbf{NC}^1$*.*

Proof. We may assume that all gates have a fan-in of at most two. Since $\mathcal{F}(\mathcal{A}(w))$ is finite, $\mathcal{F}(\mathcal{A}(w))$-$\mathbf{NC}^1 \subseteq \mathbf{NC}^1$ follows from Theorem 87 and 108. $\square$

10.5 Courcelle's Theorem

Courcelle's Theorem [Cou90] constituted a class of so-called meta theorems. It makes a claim concerning the complexity of the word problem if a restriction in the input set is imposed. In particular, given an MSO formula over graphs then Courcelle's Theorem states that it is decidable in linear time whether a graph is a model for the formula if we only consider graphs of some bounded tree-width. The generality of the theorem stems from the fact that many relevant problems are expressible in MSO.

The algorithm entails to following steps. First, a tree-decomposition has to be computed and secondly the formula has to be fitted to tree-decompositions. Checking an MSO formula on trees is then in $\mathbf{NC}^1$. In [EJT10] the overall complexity was improved to logarithmic space. In a follow-up paper [EJT12] the authors looked at the second step more closely and analyzed the complexity under the assumption that the tree-decomposition is already given. Besides confirming the $\mathbf{NC}^1$ bound in the Boolean case they regarded as an arithmetic version: Given an MSO formula and a free second-order variable X, how many valuations are there for X that satisfy the formula? The upper bound they achieved is $\#\mathbf{NC}^1$. We will re-prove this, however, note that [EJT12] is embedded in the setting of finite model theory that is slightly more general. To keep things simple, we restrict ourselves to ordinary graphs and trees.

Theorem 110 ([EJT12])**.** *For a fixed* $w \in \mathbb{N}$ *and an MSO formula* ϕ *with one free second-order variable* X*, the problem of answering the following question is in* $\#\mathbf{NC}^1$*: Given a graph* G *as a tree-decomposition of width* $w-1$*, how many valuations* $\nu\colon X \to V(G)$ *exist such that* $G \models^{\nu} \phi$*?*

Proof. **1. step.** Consider the proof for Courcelle's Theorem. Proving it takes the following steps:

1. Compute the tree-decomposition of the input graph.
2. Compile the MSO formula into a new one that fits to tree-decompositions.
3. Check if the tree-decomposition is a model for the new MSO formula.

The first one we do not have to exercise since in our case the input already is a decomposition. So, at the beginning we are interested in the second step. The standard construction in [Cou90] results in the following: If $\phi(X)$ is an MSO formula over graphs with free second-order variable X, the corresponding new formula $\phi'(X_1, \ldots, X_w)$ over tree-decompositions has w free second-order variables. There is a correspondence between subsets of $V(G)$, i.e. valuations of X, and valuations of $X_1, \ldots, X_w$: For each $S \subseteq V(G)$ there exists exactly one corresponding $S'_1, \ldots, S'_w \subseteq V(T)$, i.e. $G \models \phi(S)$ if and only if $T \models \phi'(S'_1, \ldots, S'_w)$. Note that valuations for $S'_1, \ldots, S'_w$ must have a certain form, which is imposed by the constriction of ψ'. Valuations that are not well-formed are dismissed by the formula. By the reasoning above it follows that the number of valuations for X that satisfy $G \models \phi(X)$ is equal to the number of valuations for $X_1, \ldots, X_w$ that satisfy $T \models \phi'(X_1, \ldots, X_w)$. Hence, we only have to show that we can count the number of fulfilling valuations in the formula over the tree-decomposition.

In the following we assign formulas with free variables the semantics of accepting $\mathcal{V}$-structures. For $\mathcal{V}$-structures of words we refer to [Str94]. In our case a $\mathcal{V}$-structure is a tree that is not only labeled with Σ but also with a bit telling whether a position is in X or not; hence the alphabet then is $\Sigma \times \{0, 1\}$ or $\Sigma \times \{0, 1\}^w$ if we have several free variables respectively.

The idea then is that a formula with a free variable represents a language of $\mathcal{V}$-structures. Each input together with a valuation for the free variables translate to one $\mathcal{V}$-structure and each $\mathcal{V}$-structure belongs to a tree that we get by stripping it of the variable information. In the following we consider the language of $\mathcal{V}$-structures. Given a formula with a free variable and an input tree, we count how many $\mathcal{V}$-structures based on this tree fulfill the formula. This we will achieve using extend algebras.

Let $\phi'(X_1, \ldots, X_w)$ be the MSO formula we get from $\phi(X)$ by the standard construction of Courcelle. Let $(H; +, 0_H)$ be the horizontal monoid of the syntactic extend algebra of the tree language defined by $\phi'(X_1, \ldots, X_w)$ interpreted over $\mathcal{V}$-structures as shown above. The algebra we will use for counting is

$$\mathcal{A} = (\mathbb{N}^H; \circledast, (\oplus_a)_{a \in \Sigma}, 0).$$

An element of the domain is a function $H \to \mathbb{N}$ that holds the information of how many possibilities there are to end up with some element of H, whereas the multitude of possibilities arises through the different valuations of the free variable that is coded into the word. So, if we interpret the input tree as a term over this algebra, we get the number of valuations.

The operations of $\mathcal{A}$ are defined as follows:

- The constant operation $0\colon H \to \mathbb{N}$ is defined as
$$0(h) = \begin{cases} 1 \text{ if } h = 0_H \\ 0 \text{ else} \end{cases} .$$

- For $f_1, f_2 \in \mathbb{N}^H$, we let $f_1 \circledast f_2 = f$ with
$$f(h) = \sum_{h_1 + h_2 = h} f_1(h_1) f_1(h_2).$$

- For $f \in \mathbb{N}^H$ we let
$$\oplus_a(f)(h) = \sum_{\triangle_a(h') = h} f(h'),$$
where $\triangle_a$ is an extend operation of the extend algebra.

The algebra $\mathcal{A}$ has the same signature as the syntactic extend algebra, so we can directly evaluate the term over $\mathcal{A}$. As a result we get a map that tells us for each element of H how many ways there are to obtain it. If we sum all values that correspond to elements of the accepting subset of H, we have the final output.

2. step. The algebra $\mathcal{F}(\mathcal{A})$ has the domains $\mathbb{D} = \mathbb{N}^H$ and $\widetilde{\mathbb{D}} \subseteq (\mathbb{N}^H)^{\mathbb{N}^H}$. We code $c(\mathbb{N}^H) = \mathbb{N}^n$ where $n = |H|$. Since we only use addition and multiplication the consequence is that we can represent the elements of $\widetilde{\mathbb{D}}$ as functions of the form $x \mapsto xA + b$ where A is a matrix and b is a vector. Hence, $c(\widetilde{\mathbb{D}}) = \mathbb{N}^{n \times n} \times \mathbb{N}^n$. This conforms with the operations of the algebra:

- The operations of $\mathcal{A}$ translate straight forward to the coded versions: $\circledast^c$, $\oplus_a^c$ for $a \in \Sigma$, and 0^c.

- $\circ^c$: Given $f, g \in \widetilde{\mathbb{D}}$ with $c(f)\colon x \mapsto xA_1 + b_1$ and $c(g)\colon x \mapsto xA_2 + b_2$ we have that $c(f \circ g) = c(f) \circ^c c(g)$ is a map $x \mapsto (xA_2 + b_2)A_1 + b_1 = xA_2A_1 + b_2A_1 + b_1$, so $c(f) \circ^c c(g) = (A_2A_1, b_2A_1 + b_1)$.

- $\odot^c$: Given a function $f \in \widetilde{\mathbb{D}}$ with $c(f)\colon xA + b$ and a vector $c(d) \in \mathbb{N}^n$ we have $c(f \odot d) = c(f(d)) = c(f) \odot^c c(d) = x \mapsto dA + b$.

- $\widetilde{\circledast}^c$: Given a function $f \in \widetilde{\mathbb{D}}$ with $c(f)\colon xA + b$ and a vector $d \in \mathbb{N}^n$ we have that $c(f \widetilde{\circledast} d) = c(f) \widetilde{\circledast}^c c(d)$ is of the form $x \mapsto xAM_d + bM_d$ where M_d is a matrix where position (i,j) has value $\sum_{h_i = h_j h} d_h$ where $h_i, h_j \in H$ are the elements corresponding to vector positions i and j and d_h is the value of d representing h.

- $\widetilde{\oplus}_a^c$: Given a function $f \in \widetilde{\mathbb{D}}$ with $c(f)\colon xA + b$ we have that $c(\oplus_a f) = \oplus_a^c c(f)$ is the map $x \mapsto xAM_a + bM_a$ where M_a is a matrix where position (i,j) is 1 if $\oplus_a(h_i) = h_j$ where $h_i, h_j \in H$ are the elements corresponding to vector positions i and j. In all other positions M_a is 0.

3. step. All operations are performed on matrices and vectors of a fixed size with natural values. Therefore, we can implement them in $\#\mathbf{NC}^0_{\mathbb{N}}$, which yields the overall complexity of $\#\mathbf{NC}^1$. □

Since $\#\mathbf{NC}^1$ is a subset of logarithmic space, a consequence is that MSO-counting problems on bounded tree-width graphs are also in logarithmic space.

10.6 NP-Complete Problems Parameterized by NLC-Width

In this section we look at two examples of Karp's classical 21 **NP**-complete problems [Kar72]:

- Finding maximal cuts in graphs
- Finding Hamiltonian circuits in graphs

Of course, since those problems are **NP**-complete we can hardly hope for finding a parallel algorithm in general. However, by limiting to certain inputs, we can do so indeed. We will focus on graphs of bounded NLC-width as a limitation. This is equivalent to bounded clique-width and a generalization of bounded tree-width.

We will generalize the problems in that way that we count how many solutions there are, i.e. how many maximal cuts or how many Hamiltonian circuits a graph has.

We presume that the inputs are already decomposed graphs. The best known upper bound for finding a decomposition in the case of bounded width is **P** [OS06]. Since we show lower complexity for solving the problems on the decompositions, actually finding the decompositions is the computational bottle neck. If the inputs get further restricted to bounded tree-width, then this is not the case any more since a tree-decomposition can be found in logarithmic space [EJT10].

Both problems are very similar to prove and the proof idea originates in [Wan94]. There, a **P** bound for both is shown, which we improve to $\mathbf{SAC}^1$. In the counting case we get $\#\mathbf{SAC}^1$. For the problem of counting Hamiltonian cycles, this was already discovered recently [BDG15].

Formally, the two problems are defined as follows:

Definition 111 (Maximal cuts in decomposed graphs)**.** *The input is an NLC-decomposition of an undirected graph $G = (V; E)$ of width k. Now, let $V_1 \cup V_2$ be a partition of V, which we call a cut and let $|\{\{e_1, e_2\} \in E \mid e_1 \in V_1 \wedge e_2 \in V_2\}|$ the value of the cut. The output is the value of a maximal cut and in the counting version also the number of those maximal cuts.*

Definition 112 (Hamiltonian cycles in decomposed graphs)**.** *The input is an NLC-decomposition of an undirected graph $G = (V; E)$ of width k. The output is one bit indicating whether G has a Hamiltonian cycle. In the counting version the output is the number of Hamiltonian cycles in G.*

The proof idea in both cases is the same. The input is a decomposition, so it can be considered to be a term that evaluates to the original graph. What we do now is that we evaluate this term over a different algebra that is designed to capture either cuts or cycles. Evaluating this algebra then solves the problem. This key idea comes from [Wan94] which implements it for the Boolean case. Our contribution lies in showing that this term can be evaluated in parallel, even for the counting case, following the template we presented.

Recall that the definition of NLC-decomposition consisted of three rules. These rules become the three operations of an algebra for building the graph of with k:

$$\big(\mathbb{D}; (\otimes_l)_{l\colon [k]\to[k]}, (\circledast_S)_{S\subseteq[k]\times[k]}, (\dagger_i)_{i\in[k]}\big)$$

Here, let $\mathbb{D}$ be the set of all k-colored graphs, $\dagger_i$ is a 0-ary operation, which is a graph with a single node colored i, $\otimes_l\colon \mathbb{D} \to \mathbb{D}$ recolors the graph according to l, and $\circledast_S\colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ connects two graphs via S. In the following we will alter $\mathbb{D}$ and change the semantic of the operations accordingly.

The notion of a *singleton property* of functions will be a useful tool in the following proofs. First, consider the Boolean case, and thus functions that map sets to sets, so $f\colon 2^{\mathcal{X}} \to 2^{\mathcal{X}}$. We say that f has singleton property if $f(X) = \bigcup_{x\in X} f(\{x\})$. Note that the set of functions $2^{\mathcal{X}} \to 2^{\mathcal{X}}$ has a larger cardinality than the set of functions that have singleton property. To store such a function, we only need to remember $f(\{x\})$ for all $x \in \mathcal{X}$.

The singleton property can be generalized. Consider the set of functions of the form $\mathbb{N}^{\mathcal{X}} \to \mathbb{N}^{\mathcal{X}}$. Now, singleton property means that for $\phi\colon \mathcal{X} \to \mathbb{N}$ we have

$$f(\phi) = \sum_{x \in \mathcal{X}} \phi(x) f(\chi_{\{x\}}).$$

Here, the sum and product is pointwise over the function. If a function satisfies this property, it is already defined through its image of the singletons $f(\chi_{\{x\}})$.

It would be possible to generalize the singleton property even further to other rings than $(\mathbb{N}; +, \times)$.

Theorem 113. *Counting the number of maximal cuts in graphs of bounded NLC-width is in* $\#\mathbf{SAC}^1$*.*

Proof. **1. step.** We are given the NLC-decomposition of a graph G, which is a term over the algebra defined above. We look for maximal cuts, which are partitions of $V(G)$ into V_1 and V_2 as defined above.

We now define a different algebra that, if the term is evaluated over, yields the desired value. Here, let $\mathcal{X} = [n]^{2k+1}$. In fact, we define a family of algebras $(\mathcal{A}_n)_{n \in \mathbb{N}}$ with

$$\mathcal{A}_n = \left(\mathbb{N}^{\mathcal{X}}; (\otimes_l)_{l\colon [k] \to [k]}, (\circledast_S)_{S \subseteq [k] \times [k]}, (\dagger_i)_{i \in [k]}\right),$$

where n is the number of nodes of G and k is the width. If we considered the Boolean version of the problem, we would choose $\mathcal{P}(\mathcal{X})$ for the domain and then an element is a set of vectors of the form $(a_1, \ldots, a_k, b_1, \ldots, b_k, c)$. In $\mathbb{N}^{\mathcal{X}}$ however, we store how ofter a tuple is presented instead.

The intuition behind the vectors this is that they correspond to all cuts $V_1 \cup V_2$ such that each a_i is the number of elements of V_1 are labeled i, each b_i the number of elements of V_2 that are labeled i, and c is the value of the cut .

We define the operations based on [Wan94] but incorporate the counting:

- An operation $\dagger_i$ for $i \in [k]$ is 0-ary and the characteristic function of the set that contains one tuple corresponding to the graph having one node that is colored i.

- For each total map $l\colon [k] \to [k]$ we define the unary operation $\otimes'_l\colon [n]^{2k+1} \to [n]^{2k+1}$ with $(a_1, \ldots, a_k, b_1, \ldots, b_k, c) \mapsto (a'_1, \ldots, a'_k, b'_1, \ldots, b'_k, c)$ where $a'_i = \sum_{j \in l^{-1}(i)} a_j$ and $b'_i = \sum_{j \in l^{-1}(i)} b_j$. Then $\otimes_l\colon \mathbb{N}^{\mathcal{X}} \to \mathbb{N}^{\mathcal{X}}$ is derived from $\otimes'_l$. Let $f \in \mathbb{N}^{\mathcal{X}}$. Then for $u \in [n]^{2k+1}$:

$$\otimes_l(f)(u) = \sum_{u = \otimes'_l(v)} f(v).$$

- For each $S \subseteq [k] \times [k]$ we define the operation $\circledast'_S\colon [n]^{2k+1} \times [n]^{2k+1} \to [n]^{2k+1}$. Let $x = (a'_1, \dots a'_k, b'_1, \dots b'_k, c)$, $y = (a''_1, \dots a''_k, b''_1, \dots b''_k, c)$ and $x \circledast'_S y = (a_1, \dots a_k, b_1, \dots b_k, c)$. Then $a_i = a'_i + a''_i$ and $b_i = b'_i + b''_i$. Further, $c = c' + c'' + \sum_{(i,j \in S)} \left(a'_i \cdot b''_j + b'_i \cdot a''_j\right)$. Now, for $u \in [n]^{2k+1}$:

$$(f \circledast_S g)(u) = \sum_{v_1 \circledast'_S v_2 = u} f(v_1)g(v_2).$$

The evaluation of this term yields the desired value.

2. step. We first give a coding for $\mathcal{A}_n$ and then extend it to $\mathcal{F}(\mathcal{A}_n)$. Consider the domain $\mathbb{D}$ of $\mathcal{A}$: It consists of functions of the form $\mathcal{X} \to \mathbb{N}$. We can store them explicitly as a table of natural numbers:

$$c(\mathbb{D}) = \mathbb{N}^{n^{2k+1}}$$

Now, in $\mathcal{F}(\mathcal{A}_n)$ we also have the subdomain $\widetilde{\mathbb{D}}$, which contains functions of the form

$$\mathbb{N}^{\mathcal{X}} \to \mathbb{N}^{\mathcal{X}},$$

or if we apply the previous coding this is equivalent to

$$\mathbb{N}^{n^{2k+1}} \to \mathbb{N}^{n^{2k+1}}.$$

This set is uncountable, and thus too big to be coded, however, $\widetilde{\mathbb{D}}$ does not contain all functions of that form, which enables us to code them. At this point the singleton property comes into play. We will show that the functions possess it, and so it is possible to store them by a finite set of natural numbers. To do so, we perform a induction over the operations of the algebra, where the identity function is the base case. It is $\sum_{x \in \mathcal{X}} d(x)\mathrm{id}(\chi_{\{x\}}) = \sum_{x \in \mathcal{X}} d(x)\chi_{\{x\}} = \mathrm{id}(d)$, so id has singleton property.

Now let $f, g\colon \mathbb{N}^{\mathcal{X}} \to \mathbb{N}^{\mathcal{X}}$ be some function with singleton property and $d \in \mathbb{N}^{\mathcal{X}}$. Then $f \circ g$ has it also:

$$\begin{aligned}(f \circ g)(d) = f(g(d)) &= f(\sum_{x \in \mathcal{X}} g(d(x)\chi_{\{x\}})) \\ &= \sum_{x \in \mathcal{X}} f(g(d(x)\chi_{\{x\}})) \\ &= \sum_{x \in \mathcal{X}} d(x) f(g(\chi_{\{x\}})) \\ &= \sum_{x \in \mathcal{X}} d(x)(f \circ g)(\chi_{\{x\}}).\end{aligned}$$

Next we see that $\widetilde{\otimes}_l(f)$ also has singleton property:

$$\begin{aligned}(\widetilde{\otimes}_l f)(d) = \otimes_l f(d) = \otimes_l \sum_{x\in\mathcal{X}} f(d(x)\chi_{\{x\}}) \\ = \sum_{x\in\mathcal{X}} \otimes_l f(d(x)\chi_{\{x\}}) \\ = \sum_{x\in\mathcal{X}} d(x) \otimes_l f(\chi_{\{x\}})\end{aligned}$$

Let in addition $e \in \mathbb{N}^{\mathcal{X}}$, then $f\widetilde{\circledast}_S e$ has also singleton property: Consider, how $(f\widetilde{\circledast}_S e)(d)$ is computed:

$$(f\widetilde{\circledast}_S e)(d) = f(d) \circledast_S e = \sum_{x\in\mathcal{X}} d(x)f(\chi_{\{x\}}) \circledast_S e$$

The result is a function $\mathcal{X} \to \mathbb{N}$. The function $f(\chi_{\{x\}})$ is also. From now on $f(\chi_{\{x\}})$ will be denoted as f_x. So, we went from $f\colon (\mathcal{X} \to \mathbb{N}) \to (\mathcal{X} \to \mathbb{N})$ to $f_x\colon \mathcal{X} \to \mathbb{N}$. One can define an orthogonal function $f^x\colon \mathcal{X} \to \mathbb{N}$. By interpreting f_x as a vector and writing all f_x one below another for all $x \in \mathcal{X}$, we get a $|\mathcal{X}| \times |\mathcal{X}|$-matrix. The functions f_x are the rows and f^x we define to be the columns. By f^x_y we address the entry of the matrix that is at the intersection of f^x and f_y. Observe that $(f\widetilde{\circledast}_S e)(d)(a)$ for $a \in \mathcal{X}$ is:

$$(f\widetilde{\circledast}_S e)(d)(u) = \sum_{v_1 \circledast' v_2 = u} \sum_{x\in\mathcal{X}} d(x)f^{v_1}(x)e(v_2)$$

Now, $(f\widetilde{\circledast}_S e)$ has again singleton property since we obtain the following functions: $(f\widetilde{\circledast}_S e)^x = \sum_{v_1 \circledast' v_2 = x} f^{v_1} e(v_2)$ from which we can derive $(f\widetilde{\circledast}_S e)_x$, which is $(f\widetilde{\circledast}_S e)(\chi_{\{x\}})$. Hence, we get

$$(f\widetilde{\circledast}_S e)(d) = \sum_{x\in\mathcal{X}} d(x)(f\widetilde{\circledast}_S e)(\chi_{\{x\}})$$

Since in all cases the singleton property holds, we see that we only need to store $f(\chi_{\{x\}})$ for all x to fully capture the function f. This can be stored as a matrix $\mathbb{N}^{|\mathcal{X}|,|\mathcal{X}|}$. In particular:

$$c\colon \left(\left(\mathbb{N}^{\mathcal{X}}\right)^{\mathbb{N}^{\mathcal{X}}}\right) \to \mathbb{N}^{|\mathcal{X}|,|\mathcal{X}|}$$

Following this, the coding for $\mathbb{N}^{\mathcal{X}}$ is then just a vector, which can be regarded as a word over $\mathbb{N}$. The coded operations also follow directly from the considerations above.

3. step.

Now, the complexity of the operations is $c(\mathcal{F}(\mathcal{A}_n))$ is being analyzed:

- $\otimes_l^c$. This operation takes one value d, which is a function $\mathcal{X} \to \mathbb{N}$ coded as a word of natural numbers. Then by looking at the definition of $\otimes_l$ we see that each position of $\otimes_l^c(d)$ can be computed by a single unbounded addition gate.

- $\circledast_S^c$. The implementation of this operation is similar to the previous one. Here, we get two layers where the first consists of binary multiplication gates and the second is a single unbounded addition gate.

- $\widetilde{\otimes}_l^c$. We saw that the elements of $\widetilde{\mathbb{D}}$ have singleton property and that $(\widetilde{\otimes}_l f)(d) = \otimes_l f(d) = \sum_{x \in \mathcal{X}} d(x) \otimes_l f(\chi_{\{x\}})$. Thus, we use the construction for $\otimes_l^c$. We then get a layer of $\otimes_l^c$ circuits followed by a layer of binary multiplication gates and finally a single addition gate.

- $\widetilde{\circledast}_S^c$. Recall that an element of $c(\widetilde{\mathbb{D}})$ is a matrix. We showed $(f \widetilde{\circledast}_S e)(d)(u) = \sum_{v_1 \circledast' v_2 = u} \sum_{x \in \mathbb{D}} d(x) f^{v_1}(x) e(v_2)$, which basically tells us how to compute a specific entry of the matrix that is identified by d and u. Again we see bounded multiplication gates and unbounded summation gates are needed.

- $\circ^c$. We are given two functions f and g that are coded as matrices. The constructions above lead to the following way to compute the matrix for the composition: Let $\sum_{x \in \mathcal{X}} g^x$ be the sum of the x-column in g. Then $(f \circ^c g)_y^x = f_y^x \sum_{x \in \mathcal{X}} g^x$. Once more we use bounded multiplication gates and unbounded summation gates.

- $\odot^c$. Here, we are given a matrix and a vector. The result vector is the is the sum over a pointwise vector multiplication: $(f \odot^c d)(x) = \sum_{y \in \mathcal{X}} d(y) f_y^x$.

All operations are in $\#\mathbf{SAC}^0_{\mathbb{N}}$, so the original problem is in $\#\mathbf{SAC}^1$. □

If in the proof above $\mathbb{N}$ is replaced by $\mathbb{B}$, summation by disjunction, and product by conjunction, we see that this corresponds to the Boolean version of the problem and the resulting circuit becomes Boolean, hence:

Theorem 114. *The maximal cut problem for graphs of bounded NLC-width is in* $\mathbf{SAC}^1$.

We continue with the result for counting Hamiltonian circuits whose proof is close to the previous one.

Theorem 115 ([BDG15])**.** *Counting the number of Hamiltonian circuits in graphs of bounded NLC-width is in* $\#\mathbf{SAC}^1$.

Proof. **1. step.** As in the case of the maximum cut problem, we are given a tree-decomposition as a term and we assign an algebra to it such that the evaluation yields the desired result. We use a similar algebra as for counting maximal cuts, but this time we choose $\mathcal{X} = [n]^{k(k+1)/2}$ and adjust the operations accordingly. This algebra is rooted in the construction for the Boolean version in [Wan94] where $\mathcal{P}(\mathcal{X})$ is used as a domain. Instead of holding the information whether a tuple is in a set, we count how often is has been occurring, so our domain is $\mathbb{N}^{\mathcal{X}}$. Now, an element of $\mathcal{X}$ corresponds to a subset of the edges covering the vertices. We can understand this as a path coverage of V. There are many paths and each vertex is present in exactly one. The information the tuple actually holds is how many such paths go between two colors. See [Wan94] for further details. The domain we chose counts how many such path coverings result in a certain tuple.

The operations of the algebra are defined as follows.

- The 0-ary operation $\dagger_i$ is the characteristic function of the set containing the single tuple corresponding to a graph with a single node colored i.
- For each total map $l\colon [k] \to [k]$ there is a unary operation

$$\otimes_l\colon \mathbb{N}^{\mathcal{X}} \to \mathbb{N}^{\mathcal{X}}$$

that is defined using a unary operation $\otimes'\colon \mathcal{X} \to \mathcal{X}$, which is defined next. Here, we use the notation $v_{(i,j)}$ for $v \in \mathcal{X}$ and $i, j \in [k]$, which addresses a position in the vector v that is given by a bijection between the set of positions in the vector $k(k+1)/2$ and $\binom{[k]}{2}$. He have:

$$\otimes'_l(v)_{(i,j)} = \sum_{i=l(i')} \sum_{j=l(j')} v_{(i',j')}$$

as defined in [Wan94]. Now,

$$\otimes_l(f)(u) = \sum_{u=\otimes'_l(v)} f(v).$$

- For each $S \subseteq [k] \to [k]$ there is an operation

  $$\circledast_S \colon \mathbb{N}^{\mathcal{X}} \times \mathbb{N}^{\mathcal{X}} \to \mathbb{N}^{\mathcal{X}}.$$

  This operation is a counting version of the corresponding operation described in [Wan94]. There, it is defined via a procedure, which generates new elements based on present elements. In our case we additionally have to keep track of the count of paths generating a certain element. Given two vectors $v_1, v_2 \in \mathcal{X}$, a new set of vectors is generated. This is done by defining tuples (A, B, C), the initial tuple being $(v_1, 0, v_2)$. See [Wan94] for the detailed procedure.

  We want to define $(f \circledast_S g)(v)$ for all $v \in \mathcal{X}$ and define a procedure, which yields the value. First, assume the values $(f \circledast_S g)(v)$ to be 0 for all v. Then for all $v_1, v_2 \in \mathcal{X}$ do the steps of [Wan94] for generating a new set of tuples. In each step one new edge is drawn. That way, we get a DAG that originates in $(v_1, 0, v_2)$. Actually we are only interested in a spanning tree, which we get by imposing an order of the elements of S we process. We assign each triple (A, B, C) a number $\#(A, B, C)$. The initial triple $(v_1, 0, v_2)$ is assigned $f(v_1)g(v_2)$. Suppose that we now get from triple (A, B, C) to (A', B', C') in one step. Then $\#(A', B', C') = p \cdot \#(A, B, C)$ where p is the number of possibilities to draw an edge; p is fixed by (A, B, C). Each triple can be made into an element $v \in \mathcal{X}$ as seen in [Wan94]. Let $\#(v, v_1, v_2) = \#(A, B, C)$ where v_1 and v_2 are the origins of (A, B, C) and v is the vector we get from (A, B, C). Now,

  $$(f \circledast_S g)(v) = \sum_{v_1, v_2 \in \mathcal{X}} \#(v, v_1, v_2).$$

  In this sum, every summand has the factor $f(v_1)g(v_2)$ since we can combine every path covering in f, which leads to the tuple v_1 with all of g, which leads to v_2. Then this is multiplied with the number of ways we can draw edges between the two graphs.

For obtaining the Hamilton paths we have to give the last $\circledast_S$ operation (the root of the term) a special treatment. We generate the triples and then, as described in [Wan94], if the situation occurs that a triple (A, B, C) has A and C, which only consist of 0 and B has exactly one value that is non-zero then, if S indicates that we can close the loop, we have found a path. That means this would then result in a triple all zero. Now, in our counting setting we sum over all those zero-triples generated in that way, and so we get the final result.

2. step. This step is identical to the second step of the previous proof and hence we use the same coding with the only difference that $\mathcal{X}$ is different. This, however, does not impact the reasoning.

3. step. For this step we again refer to the previous proof. The complexity analysis of $\circ^c$ and $\odot^c$ is the same. The same holds for $\otimes^c$ and hence $\widetilde{\otimes}^c$. Assuming we have $\circledast^c_S$, $\widetilde{\circledast}^c_S$ follows also. Hence, we are left with $\circledast^c_S$.

We want to compute $c(f) \circledast^c_S c(g) = c(f \circledast_S g)$ for $f, g\colon [n^{k(k+1)/2}] \to \mathbb{N}$. This is a sequence of naturals and the position corresponding to v is

$$(f \circledast_S g)(v) = \sum_{v_1, v_2 \in \mathcal{X}} \#(v, v_1, v_2).$$

So, given v, v_1, v_2 we basically have to compute $\#(v, v_1, v_2)$. Keep in mind how we defined $\#(v, v_1, v_2)$ by constructing a tree of triples (A, B, C). This tree has at most depth nk^2. By adjusting the construction we can get a tree of depth k^2 by choosing the number edges for a certain pair of S in parallel. Instead of investing one step in depth for every singe edge. All edges that correspond to one pair of S are inserted at once. The corresponding number $\#(A, B, C)$ consists of factors $f(v_1)$, $g(v_2)$ and ones we get for each edge in the tree. These factors can be hard-coded. By then picking the right number we obtain $\#(v, v_1, v_2)$ and can do the summation $\sum_{v_1, v_2 \in \mathcal{X}} \#(v, v_1, v_2)$. As the depth of the trees we construct is constant in n we need only bounded fan-in multiplication gates. Further, we need an unbounded addition gate. This gives us a $\#\mathbf{SAC}^0_{\mathbb{N}}$ bound for $\circledast^c_S$.

All operations are in the bounds of $\#\mathbf{SAC}^0_{\mathbb{N}}$, so the original problem is in $\#\mathbf{SAC}^1$. □

For the same reasons we derived Theorem 114 from Theorem 113, we may formulate a Boolean version based on the previous proof:

Theorem 116 ([Wan94])**.** *The Hamiltonian circuit problem for graphs of bounded NLC-width is in* $\mathbf{SAC}^1$.

Up until now, we considered clique-width. This kind of width is computationally harder than tree-width, i.e. finding a decomposition for clique-width has an upper bound of polynomial time whereas in the case of tree-width we have logarithmic space. Since the latter is a subset of $\mathbf{SAC}^1$, if we only regard bounded tree-width, we may formulate the theorems of this sections is a way that does not require an already decomposed graph as input.

10.7 Conclusion

Summary

We began by showing a structured approach for using the evaluation algorithm in order to obtain upper bounds. This template then is applied to a wide range of problems:

- The Boolean formula value problem and evaluating arithmetic formulas, as well as evaluating over finite or distributive algebras in general.
- Problems for Boolean and quantitative automata.
- The complexity of languages recognized by circuits of polynomial size and bounded tree-width.
- Courcelle's Theorem.
- **NP**-complete problems under bounded clique-width.

This list of applications shows how widely applicable the template is. Also, all the proofs are indeed very uniform.

Contributions

The template we formulated to obtain simple uniform proofs for upper bounds is new. We already published an earlier version of it in [KLL17a, KLL17b], however, the present version is been simplified considerably.

The list of applications intended to show the utility of the template consists of new proofs of known results. Most of them we already included in [KLL17a, KLL17b]. The presentation in this thesis, however, is improved.

Besides, we not only re-proved known results but also enhanced existing ones. Those instances are the following:

- Counting accepting runs in non-deterministic VPA when the automaton is part of the input, i.e. the counting version of the uniform membership problem for VPAs, is in $\#\mathbf{SAC}^1$.
- Functions recognized by weighted VPA over a ring R are in R-$\mathbf{NC}^1$.
- Functions implemented by polynomially bounded CVPAs over the free monoid are in $\mathbf{TC}^1$.

- Polynomially bounded functions of CVPAs over an algebra $\mathcal{A}$ are in $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1 \cup \mathbf{TC}^1$. In the case of $\mathcal{A} = (\mathbb{Z}, \times, +)$ we get an upper bound of $\mathbf{TC}^1$.
- Functions of CVPAs over $(\mathbb{Z}, +)$ are in Gap$\mathbf{NC}^1$.
- Functions of CCVPAs over the free monoid are in $\mathbf{NC}^1$ and for an algebra $\mathcal{A}$ in general we get $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$.
- Functions of CCVPA over $(\mathbb{Z}, \times, +)$ are in $\#\mathbf{NC}^1$.
- Circuit families of tree-width w over an arbitrary algebra $\mathcal{A}$ recognize functions in $\mathcal{F}(\mathcal{A}(w))$-$\mathbf{NC}^1$.
- Counting the number of maximal cuts in graphs of bounded NLC-width is in $\#\mathbf{SAC}^1$.

In summary, the significance of the contributions becomes clear if one compares the size of this chapter with the size of material needed to prove those results initially.

Sources and Related Work

The complexity of evaluating Boolean and arithmetic terms was analyzed by Buss et al. in [Bus87, Bus93, BCGR92]. The complexity of VPAs was determined by Dymond in [Dym88]. Word problems for tree automata were considered by Lohrey in [Loh01]. He also looked at the uniform membership problem in which the automaton is part of the input. We picked up the idea to examine this problem and applied it to other automaton models as well. The complexity of counting accepting computations in non-deterministic VPAs was analyzed in [KLM12] and the complexity of CRA in [AM15, AKM17]. We introduced CVPAs in [KLL16]. This paper also contained a first complexity analysis, however without having the framework, which was an obstacle. The result about bounded tree-width circuits can be found in [JS14]. Courcelle's Theorem [Cou90] initially only stated a linear time bound. In [EJT10] and [EJT12] Elberfeld et al. improved on this. The **NP** problems we looked at under bounded clique-width were initially placed in **P** by Wanke in [Wan94] and in [BDG15] Balaji et al. improved the result for Hamiltonian cycles; they also considered the counting variant of the problem.

Further Research

The main goal is clear: There should be many more cases where our framework applies. As a rule of thumb, all problems that are in some way tree-structured are worth to be looked at from the perspective of term evaluation. One big class of such problems are graph problems under some bounded width assumption. If such a problem is in $\mathbf{P}$, there is a good chance that it can be placed in $\mathbf{NC}$ by our approach.

All applications we considered led to an upper bound in terms of logarithmic depth. It would be interesting to find an example for a problem that results in proper polylogarithmic depth, e.g. $\mathbf{NC}^2$. Even an artificial example would be interesting. Or, conversely, is there a deeper reason for the lack of such examples?

With regard to our list of applications we did not exhaust every single possibility; there are still some cases left to prove that should be in reach:

- We have not examined the uniform membership problem for weighted and cost register VPAs. Although routine, we did not fix the obvious variants of our results for tree and nested word automata. Both could be addressed.
- It would be very interesting to see whether it is possible to enhance the $\mathbf{TC}^1$ bound for polynomially bounded CVPAs over free monoids. This in turn would improve the bound for polynomially bounded CVPAs in general.
- For the circuit results we only looked at the Boolean case. In [JS14] also arithmetic circuits were considered. In this case it was needed to force a bound on the degree of represented polynomial. It should be possible to also reprove this result. Further, this should apply to all algebras, hence one could try to prove the upper bound $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$ for those cases. Also, the question remains whether the result can be lifted from bounded tree-width to bounded clique-width.
- We reproved a simplified counting variant of [EJT12]. One could exercise this in greater depth. Also, a variant where the MSO formula is part of the input could be possible. A lift to bounded clique-width is of interest also.

Chapter 11

Evaluation in Low Complexity

In the beginning of the second part of this work we derived upper bounds for evaluating terms over arbitrary algebras and indicated the relevance of the evaluation problem as it is in the heart of many seemingly unrelated problems.

Now, besides upper bounds, it is interesting to ask a somewhat reversed question: Given a complexity bound, how much can we still do within this bound with respect to a problem? A prime example are the regular languages, which are in $\mathbf{NC}^1$. Then we can ask which regular languages still belong to $\mathbf{AC}^0$. This question has a nice algebraic answer on which we will build upon later. Another example we have already seen as an application of evaluation: Finding Hamiltonian cycles in graphs is **NP**-hard, yet in $\mathbf{SAC}^1$ it is still possible to find cycles in graphs of bounded clique-width.

Now, we apply this to the evaluation problem itself and ask, how much evaluation is still possible in certain low complexity bounds. The final part of this work reflects ongoing work and, therefore, cannot give final answers. Also, we mostly focus on the instance of $\mathbf{AC}^0$ as a first example for a low complexity bound.

11.1 From Evaluation to Visibly Pushdown Languages - The Scenario of Low Complexity Evaluation

Theorem 87 states that evaluating terms over finite algebras is in $\mathbf{NC}^1$. This fact serves as a starting point. Below $\mathbf{NC}^1$ we have $\mathbf{TC}^0$ and $\mathbf{AC}^0$. The $\mathbf{TC}^0 \stackrel{?}{=} \mathbf{NC}^1$

question is very interesting and involved. We focus on $\mathbf{AC}^0$ for now, which should be an easier first target since $\mathbf{AC}^0$ is separated from $\mathbf{NC}^1$ [FSS84, Hås87]. Further, we also restrict ourselves to finite algebras for now. For infinite algebras things become a bit fuzzy: For example, if we consider the integers with plus and multiplication, we would assume that the resulting circuit is arithmetic, but [AAD00] states that $\#\mathbf{AC}^0$ is almost the same as $\mathbf{TC}^0$. So, the setting of infinite algebras quickly shifts to a $\mathbf{TC}^0$ vs. $\mathbf{NC}^1$ question.

When looking at a complexity that low, the input format becomes an issue. Within $\mathbf{NC}^1$, trees that are coded as words can be parsed easily. For example, the PNF conversion was in $\mathbf{TC}^0$ and also the Dyck language, which codes trees in in-order format is $\mathbf{TC}^0$. Below $\mathbf{TC}^0$ we cannot assume to actually be able to verify whether an input represents a valid term, not to mention evaluating it.

There are infinitely many ways to represent a term as a word. Categorizing those would be an interesting question by itself, but for now we want to keep things simple. Evaluating terms over finite algebras can be reduced to computations of visibly pushdown automata. In fact, visibly pushdown languages are even a generalization since they represent unranked trees whereas evaluating terms over finite algebras is equivalent to deciding ranked tree languages. The appeal of considering visibly pushdown languages is the similarity to regular languages. Time and time again it shows that both behave similarly. It is decidable whether a regular language is in $\mathbf{AC}^0$ [BCST92]. We want to show how to lift this result to VPLs and thereby harvest the rich toolkit developed for regular word languages.

To attack this problem, first note that $\mathbf{AC}^0$ equals first-order logic with arbitrary numerical predicates. So, we will present most arguments in terms of logic as this offers more structure than a monolithic circuit. This also enables us to draw connections to related problems. After, we approximate the problem by considering special cases of VPLs. In fact, for visibly counter languages we have a complete picture with respect to complexity modulo open complexity questions.

Before we scrutinize the problem of deciding whether a VPL is first-order definable, let us revisit the case for regular languages. Given a regular language L, we can use its syntactic monoid $\mathrm{Synt}(L)$ to find out whether L is in $\mathsf{FO}[<]$: L is in $\mathsf{FO}[<]$ if and only if $\mathrm{Synt}(L)$ is aperiodic. However, there are regular languages in $\mathsf{FO}[\mathsf{arb}] \setminus \mathsf{FO}[<]$ that can also be captured. The language L is in $\mathsf{FO}[\mathsf{arb}]$ if and only if its syntactic homomorphism η_L is quasiaperiodic. This again translates to modulo predicates, so $\mathsf{FO}[<, \equiv] = \mathsf{Reg} \cap \mathsf{FO}[\mathsf{arb}]$.

In the case of $\mathsf{VPL} \cap \mathsf{FO}[\mathsf{arb}]$ we have similar goals. We seek an algebraic characterization that is decidable. Also, we want to know a minimal set of predicates P, such that $\mathsf{VPL} \cap \mathsf{FO}[\mathsf{arb}] \subseteq \mathsf{FO}[P]$. We conjecture that in this case $\mathsf{FO}[P]$ is $\mathsf{FO}[+]$. The underlying reasons will become comprehensible later.

Actually, investigating the VPLs in $\mathsf{FO}[\mathsf{arb}]$ is just one perspective on first-order logic. We will, for example, also look into the case $\mathsf{FO}[\mathsf{arb}, \rightsquigarrow]$ where in addition to arbitrary numerical predicates also a matching predicate is present. However, this is precisely first-order logic over nested words. By considering this case, we move closer back to the tree case in which the tree structure is directly accessible.

If we ask for VPLs in $\mathsf{FO}[\mathsf{arb}]$, a major problem is to define a matching predicate within the logic. In $\mathsf{FO}[\mathsf{arb}, \rightsquigarrow]$ the matching predicate is built in, however, now it is interesting to not consider the logic together with all arbitrary predicates. Therefore, we will also consider $\mathsf{FO}[\mathsf{Reg}, \rightsquigarrow]$ and $\mathsf{FO}[<, \rightsquigarrow]$.

11.2 First-Order Definability of Visibly Pushdown Languages

This section deals with the question which visibly pushdown languages are in $\mathbf{AC}^0$ and how to decide the membership. The key idea behind the proof scheme is to split the VPL word problem into two parts. First, the input tree has to be parsed, as it is only present implicitly via call and return letters. Secondly, the tree has to be evaluated using the parsed tree.

11.2.1 Parsing the Tree Structure

Let us begin with a few examples. The language $L = \{a^n b^n \mid n \in \mathbb{N}\}$ is in $\mathsf{FO}[\mathsf{arb}]$, which can be shown by a formula that figures out the middle position and then checks that the first half consists of a's and the second one of b's. This formula makes use of the $+$ predicate. The language L^* is also in $\mathsf{FO}[+]$. Here, one can find a formula that quantifies over all maximal factors of the form a^*b^* that are a member of L. The language L^* is a way of extending L in a purely horizontal way, but we can also extend it vertically and introduce branching. Consider a series of languages L_i for $i \in \mathbb{N}$ with $L_0 = \{\epsilon\}$ and $L_{i+1} = \{a^n L_1 b^n a^m L_i b^m \mid n, m \in \mathbb{N}\}$. Now, L_i is in $\mathsf{FO}[+]$ for all $i \in \mathbb{N}$ but $L_\mathbb{N} = \bigcup_{i\in\mathbb{N}} L_i$ is not. The reason is that $L_\mathbb{N}$ has a kind of arbitrary nesting similar to the Dyck language. We will try to capture this property. We propose two properties and conjecture that one of them is the right one.

Recall that we call a pair (u, v) a context if uv is well-matched. Also, recall that $\Delta(w)$ is the height of a well-matched word and that a word induces a height profile.

Definition 117 (Simple height behavior (SHB)). *A VPL L has simple height behavior if for all $m \in V_L$ that satisfy the two conditions*

- *The set $\{\Delta(u) \mid (u, v) \in \eta_L^{-1}(m)\}$ is infinite.*

- *There exist $h \in H_L$ and $m' \in V_L$ such that $m'm(h) \in \eta_L(L)$.*

it holds that for all contexts $(u,v),(w,x) \in \eta_L^{-1}(m)$ the following two conditions hold:

- $|u| = |w| \Leftrightarrow \Delta(u) = \Delta(w)$
- $|v| = |x| \Leftrightarrow \Delta(v) = \Delta(x)$

The condition saying that the set $\{\Delta(u) \mid (u,v) \in \eta_L^{-1}(m)\}$ has to be infinite, filters out elements that rather can be regarded as being finite outliers that are not vertically loopable and the condition $m'm(h) \in \eta_L(L)$ ensures m to be productive. This definition basically assigns each context an up and a down slope. So, if a VPL L has SHB there exist unique rationals $\Delta_m^{\uparrow}$ and $\Delta_m^{\downarrow}$ such that for $(u,v) \in \eta_L^{-1}(m)$ we unambiguously have $\frac{\Delta(u)}{|u|} = \Delta_m^{\uparrow}$ and $\frac{\Delta(v)}{|v|} = \Delta_m^{\downarrow}$. The slope only holds for words that correspond to a context. If we translate that back to automata, we loop trough a state on the way up and simultaneously on the way down trough another state. Then all words that correspond to a context that go through these up and down loops have to have the same slope, but this now raises the question of what can happen while being inside the loop. How much may we diverge from the actual slope? So, if (u,v) is a context that goes through such a loop, we may ask, what are the slopes of contexts like (u',v') where u' is a prefix of u and v' a suffix of v? It turns out that they, of course, do not have to have the exact same slope, but they may not diverge too far. This is captured by the following property, which by itself is actually already equivalent to SHB.

Definition 118 (Bounded corridor)**.** *A VPL L has a bounded corridor if for all $m \in V_L$ that satisfy the two conditions*

- *The set $\{\Delta(u) \mid (u,v) \in \eta_L^{-1}(m)\}$ is infinite.*
- *There exist $h \in H_L$ and $m' \in V_L$ for which $m'm(h) \in \eta_L(L)$.*

it holds that for all contexts $(u,v) \in \eta_L^{-1}(m)$ the following two conditions are met:

- $\Delta(u') - \alpha \leq \frac{\Delta(u)}{|u|} \cdot |u'| \leq \Delta(u') + \alpha$ *for all prefixes u' of u.*
- $\Delta(v') - \alpha \leq \frac{\Delta(v)}{|v|} \cdot |v'| \leq \Delta(v') + \alpha$ *for all suffixes v' if u.*

Figure 11.1 indicates both the SHB property and the bounded corridor property.

The definition uses the slope $\frac{\Delta(u)}{|u|}$. Then $\frac{\Delta(u)}{|u|} \cdot |u'|$ is the height of the prefix u' that it should have if it stayed exactly on the slope, but we allow a constant divergence of α height steps up or down.

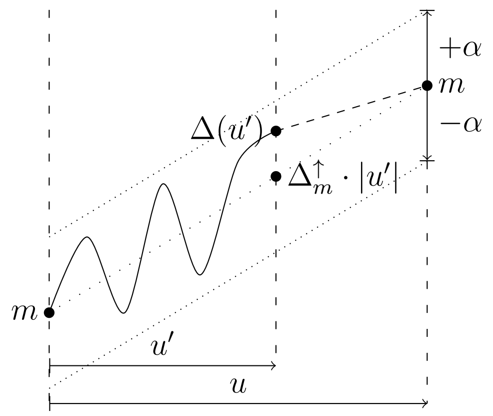


Figure 11.1: Definition 117 and 118 represent two equivalent properties that are displayed here. Both definitions share the monoid element m. The figure shows the left part of a context $(u, v) \in \eta_L^{-1}(m)$ that qualifies for the restrictions imposed by the definitions. The rational number $\Delta^{\uparrow}_m$ is the slope that can be extracted from the SHB definition.

Lemma 119. *The SHB property and the bounded corridor property are equivalent.*

Proof. Let L be some VPL. If L does not have the bounded corridor property, then for every $\alpha \in \mathbb{N}$ we find $(u, v) \in \eta_L^{-1}(m)$ for some $m \in V_L$ following the conditions imposed in Definition 118 such that there exists a prefix u' of u for which $\Delta(u') \notin \left[\frac{\Delta(u)}{|u|} \cdot |u'| - \alpha, \frac{\Delta(u)}{|u|} \cdot |u'| + \alpha\right]$ or a suffix v' of v with a similar property. We exercise the proof only for the u case. Now, for every $n \in \mathbb{N}$ there exists α such that u has a factor y that is well-matched and has a height profile that exceeds n; let $u = xyz$. By using the context-free pumping lemma we see that there is a partition of y into $y = y_1y_2y_3y_4y_5$ for which $y \sim_L y_1y_2^iy_3y_4^iy_5$ for all $i \in \mathbb{N}$. Now, we have $u \sim_L xy_1y_2^iy_3y_4^iy_5z$ where $\Delta(u) = \Delta(xy_1y_2^iy_3y_4^iy_5z)$ and see that both words have a different length, so SHB is violated.

On the other hand, if L does not have SHB, we can show that L also does not have the bounded corridor property. So, consider $(u_1, v_1), (u_2, v_2) \in \eta_L^{-1}(m)$ for some $m \in V_L$ following the conditions imposed in Definition 118 such that $u_1 = u_2$ but $\Delta(u_1) < \Delta(u_2)$. If such an m exists, we also find such an element that is idempotent. We just presume now m to be idempotent. For each $n \in \mathbb{N}$ the contexts $(u_1u_2)^n$ and $(v_2v_1)^n$ are in $\eta_L^{-1}(m)$. No matter how a corridor $\alpha \in \mathbb{N}$ is chosen, we find an $n \in \mathbb{N}$ such that $u_1^nu_2^n$ does not stay within the α-corridor; we choose the prefix u_1 to see that. Note that $\frac{\Delta(u_1^nu_2^n)}{|u_1^nu_2^n|} = \frac{\Delta(u_1u_2)}{|u_1u_2|}$ and $\Delta(u_1^n) = n\Delta(u_1)$. So, we have to show that

$$\Delta(u_1^n) \notin \left[\frac{\Delta(u_1u_2)}{|u_1u_2|} \cdot |u_1^n| - \alpha, \frac{\Delta(u_1u_2)}{|u_1u_2|} \cdot |u_1^n| + \alpha\right].$$

In particular, we want to show that we can choose n such that we fall below the lower border of the interval because we assumed $\Delta(u_1) < \Delta(u_2)$. Thus, we have:

$$\begin{aligned} \Delta(u_1^n) &< \frac{\Delta(u_1u_2)}{|u_1u_2|} \cdot |u_1^n| - \alpha \\ \Leftrightarrow \qquad \alpha &< n\left(\frac{\Delta(u_1u_2)}{2} - \Delta(u_1)\right) \end{aligned}$$

So, we find an $n \in \mathbb{N}$ if $\frac{\Delta(u_1u_2)}{2} > \Delta(u_1)$, which is true. □

Besides SHB there is another less restrictive property one can formulate that captures good height behavior:

Definition 120 (Weak SHB property (WSHB))**.** *A VPL L has weak SHB property if there exists no word $\alpha z\beta \in L$ with $z = uvwxy$ such that v, x and z are well-matched, not in Σ_{int}^*, and $z \sim_L v \sim_L x$.*

The name WSHB is justified by the following lemma:

Lemma 121. *Given a VPL L, then if L has the SHB property, it also has the WSHB property.*

Proof. Suppose that L does not have WSHB. Then there exists a well-matched word $z = uvwxy$ with $z \sim_L v \sim_L x$. Observe that there exists $m \in V_L$ for which both (uvw, y) and $(uzvw, y)$ are in $\eta_L^{-1}(m)$. This means that $\Delta(uvw) = \Delta(uzvw)$ but $|uvw| \neq |uzvw|$, which violates the SHB property. □

There is an equivalent characterization of WSHB, which is given in terms of a graph property. Let $F = (V; E)$ be some forest and $c\colon V \to [k]$ be a coloring of F. We call c a <u>branch-free coloring</u> of F if for all nodes x, y, z having the same color it holds that if x is an ancestor of y and z then either y is an ancestor of z or z is an ancestor of y.

The <u>*branch-free coloring number*</u> of a forest F is the smallest number k such that there exists a branch-free coloring $c\colon V \to [k]$ for F. For a set of forests, the branch-free coloring number is the maximum branch-free coloring number of its forests. If it does not exist, we say it is bounded. The branch-free coloring number of a VPL L is the branch-free coloring number of $\mathsf{forest}(L)$.

Lemma 122. *A VPL L has WSHB if and only if it has a bounded branch-free coloring number.*

Proof. For each $w \in L$ we consider $\mathsf{forest}(w)$. We assign the coloring $c\colon V(\mathsf{forest}(w)) \to [|H_L|]$ and assume an isomorphism between H_L and $[|H_L|]$. We assign each node v in $\mathsf{forest}(w)$ the color $\eta_L(t)$ where t is the maximal subtree that has v as its root. If the forest has WSHB, this coloring is branch-free and since H_L is finite, the coloring is bounded.

On the other hand, if WSHB does not hold, there exists a word $\alpha z\beta = \alpha uvwx\beta \in L$ such that $z \sim_L u \sim_L x$. If z needs k colors to have a branch-free coloring, then $uzwzy = uuvwxywuvwxyy$ needs $k+1$ colors. Hence, for all $k \in \mathbb{N}$ we find a word in L that is not colorable using k colors.

□

For the property of having a bounded branch-free coloring number there again exists another equivalent property:

Lemma 123. *A tree T has branch-free coloring number k if and only if the depth of the deepest complete binary tree that is a minor of T is k.*

Proof. If T has a bounded branch-free coloring number k, then, as we show first, the deepest complete binary that we find as a minor in an element of $\mathsf{forest}(L)$ has depth k. To do so, we show that the complete binary tree of depth d needs at least d colors to be colored branch-free. Say, we begin coloring at the root and we have d colors to choose from. The color we used for the root may be used again in at most one of the subtrees that are rooted in the children of the root. So, at the root level, which we index by 0, we have one node with d colors to choose from. On the next level we have one node with d and one with $d-1$ colors to choose from – otherwise we violate the coloring. From there on this coloring scheme repeats and in level 2 we have nodes with d, $d-1$, and $d-2$ colors to choose from. So, in general, in level i we have at least one node that has at most $d-i$ colors to choose from. In order to color the whole graph, the number of colors has to be the depth of the graph. If we now have a bounded coloring, then the depth of the deepest binary tree has to be bounded as well.

Conversely, if we know that the largest complete binary tree minor has depth k, one can construct a coloring using k colors. First, we show how to color a general binary tree that does not contain a complete binary tree of depth $k+1$. We assign the root some of the k colors. Let d_1 and d_2 be the depths of largest complete binary subtrees contained in the left and right descendant subtrees of the root. We know that at least one of d_1 and d_2 have to be smaller than k, otherwise the whole tree would have a minor of depth $k+1$. We choose one of the descendants of the root, which may use all k colors, while the other may not use the color of the root. The choice depends on d_1 and d_2: The descendant for which d_1, or d_2 respectively, is larger may use k colors. If $d_1 = d_2$, the choice is arbitrary. We repeat this for all nodes top to bottom and get a coloring for the whole binary tree. Now back to T, which might not be binary. Here, we may have nodes that have only one descendant. In this case it receives the same color as the parent. If there are more than two descendants, we treat all of them equally, except for the one with the largest minor. □

From the previous lemma we immediately get the statement for VPLs:

Lemma 124. *A VPL L has bounded branch-free coloring number if and only if there exists $k \in \mathbb{N}$ such that the depth of the deepest complete binary tree that is a minor of a forest in* $\mathsf{forest}(L)$ *is k.*

There is yet another property that we may use to capture trees that have a limited structural complexity, which can be found in terms of the *Horton-Strahler numbers* [Cho95, Str52, Str57]. The property will turn out to be equivalent to WSHB. Horton–Strahler numbers are assigned to nodes of a forest by the following rules: All leaves are assigned the number 1. The numbers for the other nodes are defined recursively. Let v be a node, v_1 to v_n its descendants, and h_1 to h_n the Horton–Strahler numbers for v_1 to v_n. Let $i \in [n]$ be an index for which h_i is maximal. If i is unique, then the Horton–Strahler number of v is h_i. If i is not unique, this means that there exists $j \neq i$ such that $h_j = h_i$. In this case the Horton–Strahler number of v is $h_i + 1$. The number of the root then is the number that we assign to the whole tree. In a forest we take the maximum of all roots. It is known that this number then is the same as the depth of the deepest complete binary tree we can find as a minor [Neb00].

Ultimately, we want to define a matching predicate in the case we have SHB. We know that then we also have WSHB, which is beneficial in the construction of the predicate, but we need yet another equivalent property. To that end we define $\mathsf{cancel}\colon \mathrm{WM}(\hat{\Sigma}) \to \mathrm{WM}(\hat{\Sigma})$ such that $\mathsf{cancel}(w)$ is the word one gets as result if all maximal linear factors u of w, that is $u \in (\Sigma_{\text{call}} \cup \Sigma_{\text{int}})^*(\Sigma_{\text{ret}} \cup \Sigma_{\text{int}})^* \cap \mathrm{WM}(\hat{\Sigma})$, are replaced by a word $c^{|u|}$ where c is some internal letter.

Lemma 125. *A VPL L has WSHB property if and only if it holds that* $\mathsf{cancel}^{|H_L|}(w) = \Sigma_{\text{int}}^*$ *for all $w \in L$.*

Proof. To simplify things we translate the statement to forests. Then, for some forest f, $\mathsf{cancel}(f)$ just deletes paths from leaves up to the first nodes that have more than one descendant. This first node on the way up that has more than one descendants then stays. Consider some node v in the forest with descendants v_1 to v_n. Analyzing how often we have to apply cancel until v is deleted, let d_1 to d_n be the numbers we have to apply cancel such that v_1 to v_n are deleted. Assume that $d_1 \geq d_2 \geq \ldots \geq d_n$. If $d_1 > d_2$, then v disappears together with v_1 since all the siblings are already gone. If $d_1 = d_2$, then we need $d_1 + 1$ applications of cancel.

What we described are precisely the Horton-Strahler numbers and we already saw that WSHB equals to bounded Horton-Strahler numbers. We also saw the equivalence to bounded branch-free coloring and the proof of Lemma 122 gave a construction using $|H_L|$ many colors. □

In summary, given a VPL L, the following are equivalent:

- L has WSHB.
- L has a bounded branch-free coloring number.
- $\mathsf{forest}(L)$ does not have arbitrarily large complete binary trees as minors.
- $\mathsf{forest}(L)$ has a bound on its Horton-Strahler numbers.
- For all $w \in L$ it holds that $\mathsf{cancel}^{|H_L|}(w) = \Sigma_{\text{int}}^*$.

The SHB property enables us to compute the structure of a well-matched input word in $\mathsf{FO}[\mathsf{arb}]$. Of course, this cannot work for arbitrary inputs, but it is sufficient to be able to compute the structure for words of the language in question. For words that are not in the language we accept false negatives, i.e. the case that we cannot affirm every matching. So, the next step we take is defining a matching predicate $\rightsquigarrow_L$ relative to L, which gives us the matching for all words in the language L. Note that $\rightsquigarrow_L$ is actually not unique, since we do not care what is says to matching positions within words outside the language.

Proposition 126. *If a VPL L has SHB property, then there exists a $\mathsf{FO}[+]$ formula $\rightsquigarrow_L$ with two free variables x and y such that the following hold:*

- *For $w \in L$ holds that $w \models^{x=i,y=j} \rightsquigarrow_L$ if and only if $i \rightsquigarrow j$ in w.*
- *For $w \notin L$ holds that $w \models^{x=i,y=j} \rightsquigarrow_L$ implies that $i \rightsquigarrow j$ in w.*

Proof. We design the formula $\rightsquigarrow_L$ that gives us the matching for words in the language. It uses the fact that words in the languages have certain helpful properties like the one shown in Lemma 125. For words outside the language the formula may be lucky to still compute a matching if the word happens to have a height profile that is not too complicated.

To define the predicate we will use the SHB property itself, the bounded corridor property (Lemma 119) and the property of Lemma 125 which is implied by SHB. Therefore, we will use the following constants, which exist due to SHB:

- $c \in \mathbb{N}$: The corridor number, which is the maximum over all corridor sizes α for all $v \in V_L$; see Definition 118.

- $n_v^{\uparrow}/d_v^{\uparrow} \in \mathbb{Q}$: The rational number for $v \in H_L$ that is the unique slope of the up-word meaning that it is the slope for the words x for which there exists a word y such that $(x,y) \in \eta_L^{-1}(v)$. Similarly, $n_v^{\downarrow}/d_v^{\downarrow}$ is the slope of the down-words.

The definition of the predicate $\rightsquigarrow_L$ follows the idea that is implied by Lemma 125. It says that a word in the language has at most $|H_L|$ nestings of linear words. So, we define a matching predicate $M_i(x,y)$ that can handle up to i nestings, which uses $M_{i-1}(x,y)$. Here, $M_i(x,y)$ tests whether there are two positions x', y' in the word such that $(w_x \dots w_{x'-1}, w_{y'+1} \dots w_y)$ is a context. This context can be verified because of bounded corridor and fixed slope. The word $w_{x'} \dots w_{y'}$ is verified by calls of $M_{i-1}(x,y)$.

Formally this is expressed as:

$$M_i(x,y) = x < y \wedge Q_{\Sigma_{\text{call}}}(x) \wedge Q_{\Sigma_{\text{ret}}}(y) \wedge M_i'(x+1, y-1)$$

where M_i' is again a formula with two free variables:

$$\begin{aligned} M_i'(x,y) = \exists u \exists v \;\; x \le u < v \le y \wedge \bigvee_{m \in V_L} \\ &S_c^{\uparrow,m}(x, u-1, x) \wedge u - x \bmod d_m^{\uparrow} = 0 \\ &\wedge S_c^{\downarrow,m}(v+1, y, v+1) \wedge v - y \bmod d_m^{\downarrow} = 0 \\ &\wedge d_m^{\uparrow} n_m^{\downarrow}(u-x) = d_m^{\downarrow} n_m^{\uparrow}(y-v) \\ &\wedge \forall u' \;\; (Q_{\Sigma_{\text{call}}}(u') \wedge u \le u' < v) \\ &\quad \rightarrow \exists v' \;\; Q_{\Sigma_{\text{ret}}}(v') \wedge u' < v' \le v \wedge M_{i-1}(u', v') \end{aligned}$$

The recursive definition of the above formulas is initialized by the following formula for the case $i = 0$:

$$M_0'(x,y) = \forall z \;\; x \le z \le y \rightarrow Q_{\Sigma_{\text{int}}}(z)$$

We use M_i' to actually find a context at the borders of the considered interval. If we verify that the interval indeed is built in the desired way, we cannot be sure that x and y are matching positions. For example, position x could carry an internal symbol. To get an actual matching, M_i is used, which checks that the first position contains a call letter and the last position contains a return letter. This also ensures that in every step from M_i to M_{i+1} we can match at least one more height step. This is needed since not all contexts have infinitely large height profiles, and thereby the corresponding vertical monoid elements are allowed to not have a fixed slope or bounded corridor, i.e. it is one of those elements of V_L that are filtered out by the precondition in definitions 117 and 118. So, these have to be computed height step by height step. There may be at most $|V_L|$ of such height steps, otherwise we found an element of V_L that we might call *loopable*, i.e. an element $m \in V_L$ for which there exists $m' \in V_L$ such that $mm' = m$ where $\eta_L^{-1}(m')$ contains a context (u, v) with $\Delta(u) > 0$. We only choose contexts with a length that is a multiple of $d_m^{\uparrow}$, or $d_m^{\downarrow}$ respectively. This may lead to leftover height steps. We let

$$\rightsquigarrow_L (x, y) = M_p(x, y).$$

Due to the previous considerations note that p needs one factor $|V_L|$ for each linear factor canceling step and an additional factor $|V_L|$ for leftover height steps in between. Finally, we multiply by 2 to cover the lowest $|V_L|$ height steps. So, we get

$$p = 2|V_L|^2.$$

In the definition of M_i' we used $S_c^{\uparrow,m}(x, y, z)$ and $S_c^{\downarrow,m}(x, y, z)$. These predicates serve to verify the slopes of the left and right part of a context of $\eta_L^{-1}(m)$. This is done by using a similar idea as for M_i. Where M_i uses M_{i-1} to be able to detect one more nesting level, $S_i^{\uparrow,m}(x, y, z)$ makes use of $S_{i-1}^{\uparrow,m}(x, y, z)$ to increase the size of the corridor it can detect by one. Basically, $S_c^{\uparrow,m}(x, y, z)$ is very similar to a formula one would construct for the language of (not necessarily well-matched) words that have a height profile bounded by some constant. What the variable z is for will become clear in a bit.

$$\begin{aligned}S_i^{\uparrow,m}(x, y, z) =\,& z \leq x \leq y \wedge \forall v \;\; x \leq v \leq y \rightarrow P_{\text{int}}^{\uparrow,m}(z, v) \vee \exists u \;\; x \leq u \leq y \wedge \\ & S_{i-1}^{\uparrow,m}(u, v, z) \vee S_{i-1}^{\uparrow,m}(v, u, z) \\ & \vee (((P_{\text{call}}^{\uparrow,m}(v, z) \wedge P_{\text{ret}}^{\uparrow,m}(u, z)) \vee (P_{\text{call}}^{\uparrow,m}(u, z) \wedge P_{\text{ret}}^{\uparrow,m}(v, z))) \\ & \wedge ((v < u \wedge S_{i-1}^{\uparrow,m}(v + 1, u - 1, z)) \vee (v > u \wedge S_{i-1}^{\uparrow,m}(u + 1, v - 1, z))))\end{aligned}$$

The down version $S_i^{\downarrow,m}(x, y, z)$ of the previous predicate is similarly defined. The following constructions are also only exercised for the up case since the down case is symmetric.

The objective of $S_i^{\uparrow,m}$ is to verify the slope, but we did this in the same way we would check whether a word has a bounded height profile, which can be regarded as the case of slope 0. This translation is achieved by the predicates $P_{\text{call}}^{\uparrow,m}$, $P_{\text{ret}}^{\uparrow,m}$, and $P_{\text{int}}^{\uparrow,m}$. The interval we want to check is divided into chunks of a fixed length. Then the deviation from the true slope is measured and handed out. For example, if $m \in V_L$ has an up slope of 1, the word *aabaaabab* gets collapsed to *aaa* or the word *bbbbaab* becomes *bbb*. Since the corridor is bounded and 1 is the maximal slope, this translation is possible. So, using $P_{\text{call}}^{\uparrow,m}$, $P_{\text{ret}}^{\uparrow,m}$ in $S_i^{\uparrow,m}$ can be interpreted as a transduction that filters out the deviation from the slope.

The size of the blocks we check is $d_m^{\uparrow}$. The previous predicates carried the input variable z. This variable holds the offset for the blocks.

Now $w \models^{x=i,y=j} P_{\text{call}}^{\uparrow,v}(x,y)$ is defined by the property

$$\Delta\left(w_{j-(j-i \bmod d_m^{\uparrow})} \cdots w_{j-(j-i \bmod d_m^{\uparrow})+d_m^{\uparrow}-1}\right) > j-i \bmod d_m^{\uparrow}.$$

Also, $w \models^{x=i,y=j} P_{\text{ret}}^{\uparrow,v}(x,y)$ is defined by the property

$$-\Delta\left(w_{j-(j-i \bmod d_m^{\uparrow})} \cdots w_{j-(j-i \bmod d_m^{\uparrow})+d_m^{\uparrow}-1}\right) > j-i \bmod d_m^{\uparrow}.$$

Finally, we define $P_{\text{int}}^{\uparrow,v}(x,y) = \neg P_{\text{call}}^{\uparrow,v}(x,y) \wedge \neg P_{\text{ret}}^{\uparrow,v}(x,y)$.

□

Now that we defined $\rightsquigarrow_L$, recall our approach here: We split the membership problem into two parts that consist of analyzing the tree structure and then using the result to do the rest. Yet, one could interject that maybe there is a VPL that is $\mathsf{FO}[\mathsf{arb}]$-definable, but $\rightsquigarrow_L$ is not. This is actually not the case.

Lemma 127. *If a VPL L is in $\mathsf{FO}[\mathsf{arb}]$, then $\rightsquigarrow_L$ is $\mathsf{FO}[\mathsf{arb}]$-definable.*

Proof. We are given an $\mathsf{FO}[\mathsf{arb}]$ formula ϕ for L. Note that L only contains well-matched words. The predicate $\rightsquigarrow_L$ with free variables x and y ought result to true if x and y address a well-matched factor u of the input word w, which belongs to L. Let $\eta_L(u) = h$, then there exists $v_h \in V_L$ such that $v_h(h)$ is in the accepting set of the forest algebra. Let X be a set of representatives of the sets $\eta_L^{-1}(v_h)$ for all $h \in H_L$ for which an $v_h \in L$ exists as described. Now, we may construct $\rightsquigarrow_L$ with two free variables x and y. It is a disjunction over all $(u,v) \in X$ and tests for $\nu_1(x) = i$, $\nu_2(y) = j$ whether $uw_i \dots w_j v \models \phi$ for w being the input word. Now, if x and y hold two matching positions, there exists (u,v) for which $uw_i \dots w_j v \in L$. □

That the matching predicate is definable in first-order logic is, of course, not sufficient for some VPL to be first-order definable. We will come back to that later, because for now we engage in the question of when $\rightsquigarrow_L$ is definable in $\mathsf{FO}[\mathsf{arb}]$. Unfortunately we do not know whether SHB is the characterizing property. There could be languages without SHB such that $\rightsquigarrow_L$ is still definable. In the next subsections we will take a closer look at this problem. At this point we stick to the following conjecture saying that the languages we do not know the status of are not $\mathsf{FO}[\mathsf{arb}]$ definable:

Conjecture 128. *For a VPL L the predicate $\rightsquigarrow_L$ is definable in $\mathsf{FO}[\mathsf{arb}]$ if and only if L has the SHB property.*

Combining this conjecture with Lemma 127 we get:

Corollary 129. *If Conjecture 128 holds, then languages without SHB property are not in $\mathsf{FO}[\mathsf{arb}]$.*

We also formulate a weaker conjecture for matching predicates that are in $\mathsf{FO}[\mathsf{arb}]$. It is implied by conjecture 128 since we know that if L has SHB property, the matching predicate is in $\mathsf{FO}[+]$:

Conjecture 130. *Given a VPL L, if the predicate $\rightsquigarrow_L$ is definable in $\mathsf{FO}[\mathsf{arb}]$ then it is already definable in $\mathsf{FO}[+]$.*

For SHB we have the conjecture that languages that do not possess this property are not first-order describable. For WSHB we, however, can show that it is a necessary condition.

Proposition 131. *Given a VPL L without WSHB property, then L is $\mathbf{TC}^0$-hard and thereby not in $\mathsf{FO}[\mathsf{arb}]$.*

Proof. If L does not have the WSHB property, then there exists a word $z = uvwxy$ such that $z \sim_L v \sim_L x$ as described in the definition and $\alpha z \beta \in L$ for some context (α, β). We may assume $v = x$. We will use this to construct a reduction from the $\mathbf{TC}^0$-hard language $\textsc{Equality} = \{w \in \{0,1\}^* \mid |w|_0 = |w|_1\}$ to L. To that end we define a function $f\colon \{0,1\}^* \to \Sigma^*$ with

$$f(s) = \alpha u x \phi(s) \psi(s) w x y \beta$$

where ϕ and ψ are homomorphisms:

- $\phi\colon 0 \mapsto (wux)^{2|uxwy|+1}$
- $\phi\colon 1 \mapsto wux(uxw)^{2|wux|} xy^{2|wux|}$
- $\psi\colon 0, 1 \mapsto y^{|uxwy|+1}$

Given a word s, let $\bar{s}$ be the mirrored word in the sense that it is the image under a homomorphism mapping $0 \mapsto 1$ and $1 \mapsto 0$. Now, we get the reduction as follows:

$$s \in \text{EQUALITY} \Leftrightarrow f(s) \in L \wedge f(\bar{s}) \in L.$$

If a word does not have an equal number of 0's and 1's, then either $\Delta(f(s))$ or $\Delta(f(\bar{s}))$ is negative and hence not in the language. If an equal number of 0's and 1's is present, both heights are 0 and due to the constructions above, $f(s)$ and $f(\bar{s})$ are both in L. This is because $ux\phi(s)\psi(s)wxy \sim_L z$. To see that, notice that $\Delta(\phi(0)) = \Delta(wu)(2|uxwy| + 1)$ and $\Delta(\phi(1)) = \Delta(wu)$, so $\Delta(\phi(s)) = \Delta(wu)(|uxwy| + 1)|s|$ if and only if $|s|_0 = |s|_1$. Further, we have $\Delta(\psi(s)) = \Delta(y)(|uxwy| + 1)|s|$, which equals $-\Delta(\phi(s))$ if and only if $|s|_0 = |s|_1$. So, $f(s)$ is well-matched if and only if $s \in \text{EQUALITY}$. It can also be verified that $|\phi(0)| = |\phi(1)|$, which then makes the reduction computable in $\mathbf{AC}^0$. $\square$

Conjecture 132. *Given a VPL L, if $\rightsquigarrow_L$ is not in $\mathsf{FO}[\mathsf{arb}]$, then L is $\mathbf{TC}^0$-hard.*

Corollary 133. *If a VPL L is in $\mathsf{FO}[\mathsf{arb}]$, then it has WSHB property.*

11.2.2 Evaluating the Parsed Tree

We want to derive a formula for the word problem of a VPL L and have already defined a matching predicate $\rightsquigarrow_L$. To do so, we want to know what property is sufficient to place L in $\mathsf{FO}[\mathsf{arb}]$. Recall that in the case of regular languages the property is quasiaperiodicity of the syntactic homomorphism. This property we may simply lift to the VPL case.

Definition 134 (Quasiaperiodicity of VPLs)**.** *Given a VPL L, we call L quasiaperiodic if the following properties hold:*

- *For all $m \in \mathbb{N}$ the set $\eta_L(\mathrm{WM}(\hat{\Sigma}) \cap \Sigma^m)$, which is a subset of H_L, does not contain a nontrivial group.*
- *For all $m, n \in \mathbb{N}$ and all sets of contexts $X = \{(x, y) \in \Sigma^m \times \Sigma^n \mid xy \in \mathrm{WM}(\hat{\Sigma})\}$ the set $\eta_L(X)$, which is a subset of V_L, does not contain a nontrivial group.*

Actually, the second condition in the definition implies the first one. One can see this by considering the cases where $m \in \mathbb{N}$, but $n = 0$. Then X consists of contexts of the form (x, ϵ). Since $\eta_L((x, \epsilon)) \cdot' \eta_L(\epsilon) = \eta_L(x)$, the fact that $\eta_L(X)$ does not contain a nontrivial group is equivalent to the first condition. Hence, from now on, if we want to show quasiaperiodicity, we only need to show the second condition. If we want to, however, use quasiaperiodicity in some construction, we may utilize the first condition also.

Similarly to the regular case, we can show that without quasiaperiodicity, definability in first-order logic ceases.

At this point we can also define aperiodicity for VPLs: A VPL is <u>*aperiodic*</u>, if both monoids H_L and V_L are aperiodic.

Proposition 135. *A VPL L that is not quasiaperiodic is not in* $\mathsf{FO}[\mathsf{arb}]$.

Proof. We show that if L is not quasiaperiodic then we can reduce $\textsc{Mod}_p = \{w \in \{0,1\}^* \mid 0 \equiv |w_0| \pmod p\}$ to L for some prime p.

If there exists $m, n \in \mathbb{N}$ such that for $X = \{(x,y) \in \Sigma^m \times \Sigma^n \mid xy \in \mathrm{WM}(\hat{\Sigma})\}$ the set $\eta_L(X)$ contains a cyclic group $\mathbb{Z}_p$, then let $(u,v),(u',v') \in X$ for which $\eta_L(u,v)$ is the neutral element 0 of $\mathbb{Z}_p$ and $\eta_L(u',v')$ is 1 in $\mathbb{Z}_p$. Let α, β, γ be words such that $\alpha u \beta v \gamma \in L$ but $\alpha u' \beta v' \gamma \notin L$. The function f is defined as $w \mapsto \alpha\phi(w)\beta\psi(w^R)\gamma$ where w^R is the reversed word of w. Further, ϕ is a homomorphism with $0 \mapsto u$ and $1 \mapsto u'$ and ψ is a homomorphism with $0 \mapsto v$ and $1 \mapsto v'$. This mapping is an $\mathbf{AC}^0$-reduction and yields $\textsc{Mod}_p \leq L$.

$\square$

Now, on the other hand, if L has a definable matching predicate $\rightsquigarrow_L$ and is quasiaperiodic, then we can show it to be in $\mathsf{FO}[\mathsf{arb}]$. Note that a definable matching predicate implies at least WSHB, which tells us that the trees that the members of L represent have a limit in the nesting complexity, as we saw in Lemma 125. This lemma shows how to design a formula for the membership problem. The matching predicate is used to detect maximal linear words within a word. These can be evaluated within the vertical monoid, which is possible due to quasiaperiodicity. After, all newly evaluated factors are then evaluated within the horizontal monoid, which is also possible due to quasiaperiodicity. Lemma 125 tells us that we have to repeat that a fixed number of times.

So, in the approach there is a fixed number of rounds in which maximal linear factors are evaluated. This is achieved using a transduction. It evaluates linear factors and replaces them by neutral letters while one letter is containing the evaluation result. First, we show that a word that is partly evaluated is not harder to further evaluate than the original word.

Let L be some VPL over $\hat{\Sigma} = (\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \Sigma_{\text{int}})$. Then let $\bar{L}$ be a VPL over the visible alphabet $(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, H_L)$. Let $0 \in H_L$ be the neutral element. We begin with the words of L and replace every internal letter c by $\eta_L(c)$; let L' be the set of these words. Now, $\bar{L}$ is defined as the closure of L' under the following operation: Given a word $w_1w_2w_3 \in \bar{L}$, where w_2 is well-matched and neither the last letter of w_1 nor the first letter of w_3 are internal, then also

$$w_1 0^{|w_2|-i-1}\eta_L(w_2)0^i w_3 \in \bar{L}$$

for all $i \in [0, |w_2|-1]$, where for the new internal letters in H_L we assume η_L to be continued to be the identity function: $\eta_L(c) = c$ for $c \in H_L$.

Lemma 136. *The VPLs L and $\bar{L}$ are equivalent under* $\mathsf{FO}[+]$ *many-one reductions.*

Proof. First, notice that $L \leq \bar{L}$ by the function described above, which replaces each internal letter c by $\eta_L(c)$. This reduction is in $\mathsf{FO}[+]$.

For the converse, we are given a word $w \in \bar{L}$ and show that in $\mathsf{FO}[+]$ we can substitute all $0^*H_L0^*$ blocks with well-matched words that evaluate to the nontrivial element of the horizontal monoid.

We let f be the reduction with $w \in \bar{L} \Leftrightarrow f(w) \in L$.

Consider a maximal factor of the word that is an element of $0^*H_L0^*$ of length n. We want a procedure that, on an input $h \in H_L$ of length n, outputs a word w of length n such that $\eta_L(w) = h$. Note that the Parikh image of $\eta_L^{-1}(h)$ is semi-linear [Par66] and if we are only interested in word lengths A we obtain $A = \bigcup_{i\in[k]} A_i$, where $A_i = \{a_i x + b_i \mid x \in \mathbb{N}\}$ for numbers $a_i, b_i \in \mathbb{N}$. It is easy to see that each word of a sufficient length in $\mathcal{A}_i$ then is of the form uv^jwx^jy for words u, v, w, x, y depending on i and $j \in \mathbb{N}$. Finding some i such that $n \in \mathcal{A}_i$ is in $\mathsf{FO}[+]$ and then generating the word is also in $\mathsf{FO}[+]$. Short words can be treated by using a look-up-table. ☐

Proposition 137. *A VPL L is in* $\mathsf{FO}[\mathsf{arb}]$ *if and only if $\rightsquigarrow_L$ is definable in* $\mathsf{FO}[\mathsf{arb}]$ *and L is quasiaperiodic. If L is in* $\mathsf{FO}[\mathsf{arb}]$*, then it is already in* $\mathsf{FO}[+, \rightsquigarrow_L]$.

Proof. We already saw in Lemma 127 and Proposition 135 that, if $\rightsquigarrow_L$ is not in $\mathsf{FO}[\mathsf{arb}]$ or if L is not quasiaperiodic, then L is not in $\mathsf{FO}[\mathsf{arb}]$. So, for the converse we are left showing an $\mathsf{FO}[+, \rightsquigarrow_L]$ formula for L.

To construct the formula for L we will use a few transductions that are $\mathsf{FO}[+]$-computable. They extend the idea of Lemma 136. Beginning in a word $w \in L$ we transform it in a way that in the end we get a word in $0^*H_L0^*$ where the single position with a nontrivial horizontal monoid element holds the evaluation $\eta_L(w)$. Then checking whether this element is in $\eta_L(L)$ is easy. Now, we have to show that for words $w \in L$ we indeed arrive at a word in $0^*H_L0^*$ using a fixed number of transduction steps. If w is not in L, the procedure may break earlier such that we do not get an evaluation of the word and hence cannot verify that it belongs to L. So, for the rest assume that the input word is in L.

Observe that H_L has at most one absorbing element $\bot$. Due to WSHB we know that $\bot$ cannot be in the accepting set. Otherwise $\eta_L^{-1}(\bot)\mathrm{WM}(\hat{\Sigma})$ would be a subset of L that does not have WSHB. So, there is no $v \in V_L$ such that $v(\bot)$ is in the accepting set.

Let $k \in \mathbb{N}$ be some constant, then for $h \in H_L$ with $h \neq \bot$ then $Y = \eta_L^{-1}(h) \cap \Delta_k$ is in $\mathsf{FO}[\mathsf{Reg}]$, where Δ_k is the set of all well-matched words that have a height profile that does not exceed k. To show that, first note that L is regular since a finite stack can be simulated by a finite automaton. Then, if the syntactic homomorphism $\eta_Y : \Sigma^* \to \mathrm{Synt}(Y)$ is not quasiaperiodic then η_L is also not quasiaperiodic: Consider $\eta_Y(\Sigma^t)$ for $t \in \Sigma$ and assume that $x, y \in \Sigma^t$ forms a cyclic group generated by $\eta_Y(\{x, y\})$. If x and y are well-matched, then η_L is also not quasiaperiodic. The same is true for the weaker property of $\Delta(x) = \Delta(y) = 0$. In this case we find a factor in xyx that is well-matched and spans the group. We can relax the restriction even more to $\Delta(x) = -\Delta(y)$, then the same still holds. If $\Delta(x) \neq -\Delta(y)$, we get a contradiction since then we would find two syntactically equivalent words of different heights.

Now, let

$$f_k \colon \mathrm{WM}(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \Sigma_{\text{int}}) \to \mathrm{WM}(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, H_L)$$

be a transduction that replaces the largest factors $u \in \Delta_k$ by $0^{|u|-1}\eta_L(u)$. These factors can be found using k nested matching predicates $\rightsquigarrow_L$. That way the height up to k can be verified. Also, by the previous argument, we see that f_k is computable in $\mathsf{FO}[\mathsf{Reg}, \rightsquigarrow_L]$, which is a subset of $\mathsf{FO}[+, \rightsquigarrow_L]$.

The next transduction

$$g_1 \colon \mathrm{WM}(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, H_L) \to \mathrm{WM}(\Sigma_{\text{call}} \times H_L, \Sigma_{\text{ret}} \times H_L, H_L)$$

rearranges factors that are already evaluated. Given w then $g_1(w)$ is as follows: If there is a factor of the form $a0^p h 0^p a'$ for $a, a' \in \Sigma_{\text{call}}$, then the last letter a' gets transformed and the factor becomes $(a, x)0^{p+q+1}(a', h)$, where x depends on whether a receives a transformation itself; by default it is $x = 0$. If there is a factor of the form $b0^p h 0^q b'$ for $b, b' \in \Sigma_{\text{ret}}$, then this gets transformed to $(b, h)e^{p+q+1}(b', x)$. Finally, if $a0^p h e^q b$, then this becomes $(a, x)0^{p+q}h(b, x')$ where again x and x' depend on the context of the factor. The idea is that if we have a matching between two positions x and y having letters a and b, then this is a context $\eta_L((a, b))$, but this context can actually be extended if a is directly preceded by some well-matched word w and/or b is followed by some well-matched word w'. After the transformation we have all the information present in positions x and y: $(a, \eta_L(w))$ is in position x and $(b, \eta_L(w'))$ is in position y. Like f_k, the function g_1 does not make the evaluation problem computationally harder; see Lemma 136. Since g_1 just rearranges parts of the word, we get a bound of $\mathsf{FO}[<, \rightsquigarrow_L]$.

After g_1 there is another transduction

$$g_2\colon \mathrm{WM}(\Sigma_{\text{call}} \times H_L, \Sigma_{\text{ret}} \times H_L, H_L) \to \mathrm{WM}(\{\Diamond\}, \Sigma_{\text{call}} \times H_L \times \Sigma_{\text{ret}} \times H_L, H_L)$$

that, on a word w, is performing as follows: If two positions x and y are matched and have the letters (a, h_1) and (b, h_2) for $a \in \Sigma_{\text{call}}$, $b \in \Sigma_{\text{ret}}$, and $h_1, h_2 \in H_L$, then in $g_2(w)$ position x will be $\Diamond$ and position y will be (a, h_1, b, h_2). Also, this transduction is computable in $\mathsf{FO}[\rightsquigarrow]$. Now, let $g = g_2 \circ g_1$.

Before we introduce the final transduction h, note that later we want to apply all transductions multiple times. Therefore, from now on we presume them to be extended to all input alphabets we used so far in the obvious way.

Given is a word $w = ua'w_1aw_2bw_3b'v$ for uv, w_1, w_2, w_3 being well-matched, a, a' being call letters, and b, b' being return letters. Suppose that after some computation steps the letter b became (a, h_1, b, h_2). Now, it could be that h_1 already holds the value $\eta_L(w_1)$ and w_1 is replaced by $e^{|w_1|}$. It could also be that w_1 is not evaluated yet; then $h_1 = e$. Notice that $w_1 = \epsilon$ is a possibility. Similar considerations hold for h_2 and w_3. If both w_1 and w_3 are already evaluated and the evaluations are present in h_1 and h_2, we call the letter (a, h_1, b, h_2) in this position *finished*.

There is a third transduction h that evaluates the largest linear factors in which all return letters are finished. These factors can be found using the matching predicate. The evaluation is basically evaluating a word over the vertical monoid V_L. A letter (a, h_1, b, h_2) is mapped onto the context $h_1 + \eta_L((a, b)) + h_2$. Due to the vertical quasiaperiodicity condition, there is a $\mathsf{FO}[\mathsf{Reg}]$ formula for that. The transduction then replaces all positions that lie in an evaluated interval by e except the last position, which is assigned the evaluation.

To evaluate the whole word the transduction $(h \circ g \circ f_k)^{|V_L|}$ is sufficient. We only need $|V_L|$ rounds because the language has WSHB and for words in the language Lemma 125 gives us this bound.

The transduction constructed as well as the check whether the final evaluation result is part of the accepting set can be compiled into a single $\mathsf{FO}[+, \rightsquigarrow_L]$ formula. $\square$

Corollary 138. *A VPL L is in* $\mathsf{FO}[+]$ *if it has SHB property and L is quasiaperiodic. If conjecture 128 holds, the reverse is also true. In this case we have* $\mathsf{VPL} \cap \mathsf{FO}[\mathsf{arb}] \subseteq \mathsf{FO}[+]$.

11.2.3 Decidability

All properties we considered are decidable. We assume a VPL is given as a VPA or a finite forest algebra with recognizing homomorphism.

Proposition 139. *Given a VPL L, it is decidable whether L has SHB.*

Proof. The SHB property imposes a condition on certain elements of V_L. These elements can be singled out easily. For each v of them we have to verify that the slope is fixed, i.e. for $(x, y) \in \eta_L^{-1}(v)$, $\frac{\Delta(x)}{|x|}$ and $\frac{\Delta(y)}{|y|}$ are already determined by v. We do this by testing all $(x, y) \in \eta_L^{-1}(v)$ with $\Delta(x) \leq |V_L|!$. □

Proposition 140. *Given a VPL L, it is decidable whether L has WSHB.*

Proof. If L does not have WSHB, then there exists a word $w \in L$ such that $\mathsf{forest}(w)$ contains a complete binary tree of depth $|H_L|$ as a minor. Also, if such a word does not exist in L, then L does have WSHB. Let $w_h \in \mathrm{WM}(\hat{\Sigma})$ be some word of $\eta_L^{-1}(h)$ for $h \in H_L$ and let c_v be some context in $\eta_L^{-1}(v)$ for $v \in V_L$. We build well-matched words based in the complete binary tree of depth $|H_L|$. Every leaf is assigned some w_h and every edge is assigned some v_h. So, we get $|H_L|^{2^{|H_L|}}$ many possibilities for assignments to the leaves and $|V_L|^{2^{|H_L|-1}}$ possibilities for assignments to the edges. One of the words we built is in L if and only if L does not have WSHB. □

Since checking for aperiodicity in monoids is decidable we get the following:

Proposition 141. *Given a VPL L, it is decidable whether L is aperiodic.*

Proposition 142. *Given a VPL L, it is decidable whether L is quasiaperiodic.*

Proof. The quasiaperiodicity condition consist of a subcondition for H_L and V_L, however the one for V_L is already sufficient. The condition for H_L is the same as for the classical monoid case with the only difference that we only consider well-matched words. So, we do the same as in this case and test $\eta_L(\Sigma^m \cap \mathrm{WM}(\hat{\Sigma}))$ for all $m \in [|H_L|!]$. The second condition uses m and n and in this case we test all combinations for $m, n \in [|V_L|!|H_L|!]$. □

Corollary 143. *Assuming Conjecture 128 holds and given a VPL L, it is decidable whether L is in* $\mathsf{FO}[\mathsf{arb}]$.

11.3 Visibly Counter Languages

For general visibly pushdown languages there are some tricky open questions that hinder us to obtain a decidable characterization for first-order definability. However, the special case of visibly counter languages is different. Here, we have a complete picture without open questions. This is because of the following:

Proposition 144. *If a VCL L does not have SHB property, then it is* $\mathbf{TC}^0$*-hard.*

Proof. Let M be a deterministic k-VCA for L and $m \in V_L$ be such that $\eta_L^{-1}(m)$ is a witness for L not having SHB according to the definition. Now, there also exists an idempotent $m' \in V$ for which $\eta_L^{-1}(m')$ is also a witness. Using a pumping argument, one can see that there exist states $q^{\uparrow}, q^{\downarrow}$ of M and $(u, v), (u', v') \in \eta_L^{-1}(m')$ such that $q^{\uparrow} \xrightarrow{u} q^{\uparrow}$, $q^{\uparrow} \xrightarrow{u'} q^{\uparrow}$, $q^{\downarrow} \xrightarrow{v} q^{\downarrow}$, and $q^{\downarrow} \xrightarrow{v'} q^{\downarrow}$ with $|u| = |u'|$ but $\Delta(u) \neq \Delta(u')$.

Let $\alpha\gamma$ and β be well-matched words for which $\alpha u \beta v \gamma \in L$ and

$$q_I \xrightarrow{\alpha} q^{\uparrow} \xrightarrow{u} q^{\uparrow} \xrightarrow{\beta} q^{\downarrow} \xrightarrow{v} q^{\downarrow} \xrightarrow{\gamma} q_f$$

for some final state q_f.

Consider the following words:

- $x = u^{-\Delta(v)}$
- $y = u'^{-\Delta(v)}$
- $z = v^{\Delta(u)+\Delta(u')}$

We assume that $-\Delta(z) > \Delta(\alpha\beta)$, which can be achieved by powering. This is equivalent to $\Delta(\alpha\beta z) < 0$. Observe now, for example, that $\alpha xy\beta zz\gamma \in L$ since

$$\begin{aligned}\Delta(xy) &= -2\Delta(v)(\Delta(u) + (\Delta(u')) \\ &= -2(\Delta(u) + \Delta(u'))\Delta(v) \\ &= -\Delta(zz).\end{aligned}$$

The key idea is that there must be as many x's and y's in order to be able to form a word in the language using z's. This enables us to reduce the $\mathbf{TC}^0$-hard language EQUALITY to L. For the reduction we use a function $f\colon w \mapsto \alpha\phi(w)\beta\psi(w)\gamma$ where ϕ and ψ are homomorphisms with $\phi(0) = x$, $\phi(1) = y$ and $\psi(w) = z^{|w|}$. Since $|x| = |y|$, this is a $\mathbf{AC}^0$-computable function.

For some word $w \in \{0, 1\}^*$, let $\bar{w}$ be the image of w under the map that switches 0 and 1, i.e. $0 \mapsto 1$ and $1 \mapsto 0$. Now, we get

$$w \in \text{EQUALITY} \Leftrightarrow f(w) \in L \wedge f(\bar{w}) \in L.$$

This is true since if w does not have the same number of 0's and 1's then either $\Delta(f(w))$ or $\Delta(f(\bar{w}))$ is negative because $\Delta(\alpha\beta z) < 0$ and hence $f(w)$ or $f(\bar{w})$ is not in L. If w has an equal number of 0's and 1's, then both $f(w)$ or $f(\bar{w})$ are in L. □

Corollary 145. *A VCL is in* $\mathsf{FO}[\mathsf{arb}]$ *if and only if it is quasiaperiodic and has SHB. All VCLs in* $\mathsf{FO}[\mathsf{arb}]$ *are in* $\mathsf{FO}[+]$. *Moreover, it is decidable whether a VCL is in* $\mathsf{FO}[+]$.

11.4 An Open Problem

We saw two ways to capture the complexity of the height profile of words of a VPL. First, we had SHB for which we were able to show that this property leads to a first-order definable matching predicate $\rightsquigarrow_L$. We also showed that if a language is first-order definable then $\rightsquigarrow_L$ is also. So, if $\rightsquigarrow_L$ is not first-order definable, L is neither. We were unable to show that SHB is a necessary condition for first-order definability. Besides SHB we considered WSHB for which the situation is reversed. We were able to show that it is necessary, but sufficiency remains unknown.

For VCLs the situation turned out to be nicer: A VCL without SHB is not first-order definable. For the general VPL case we try to map out the vicinity of this open problem and connect the VPL and the VCL case.

Let, given a VPL L, L^Δ be the set of height behaviors, i.e.

$$L^\Delta = \{v \in \mathbb{N}^* \mid \exists w \in L \colon |w| = |v| \wedge v_i = \Delta(w_1, \ldots, w_i)\}.$$

The following lemma is immediate:

Lemma 146. *Given two VPLs L and M, if $L^\Delta = M^\Delta$, then $\rightsquigarrow_L$ is also a matching predicate for M.*

If we combine the previous lemma with Lemma 127, we get the following.

Lemma 147. *If for some VPL L the matching predicate $\rightsquigarrow_L$ is not* $\mathsf{FO}[\mathsf{arb}]$*-definable, then all languages M with $L^\Delta = M^\Delta$ are not in* $\mathsf{FO}[\mathsf{arb}]$.

One of the simplest languages that is a VCL and not in $\mathsf{FO}[\mathsf{arb}]$ because of lacking the SHB property is

$$\mathcal{L}_1 = L(S \to aSb|acSb|\epsilon).$$

For all examples we present here we give the productions of a context-free grammar where S is the initial non-terminal. One can also directly show that $\mathcal{L}_1$ is not in $\mathsf{FO}[\mathsf{arb}]$ by using a reduction of Equality which follows the idea of Proposition 131. Now, consider the language

$$\mathcal{L}_2 = L(S \to aSb_1|acSb_2|\epsilon).$$

Here, the return letters b_1 and b_2 carry the information whether or not the matching position is followed by the letter c. For this language the reduction used for $\mathcal{L}_1$ fails. However, by Lemma 147 we see that $\mathcal{L}_2$ also is not in $\mathsf{FO}[\mathsf{arb}]$. As a corollary we get:

Corollary 148. *Given a VPL L for which there exists a VCL M with $L^\Delta = M^\Delta$, it is decidable whether L is in* $\mathsf{FO}[\mathsf{arb}]$.

We conjecture that all VPLs that are in $\mathsf{FO}[\mathsf{arb}]$ have a VCL with the same height behavior. We suspect that the same holds for VPLs that have SHB. We have no proof for neither, but also no counter example. We can at least bridge the gap a bit by considering VVPLs:

Lemma 149. *For every VPL L there exists a VVPL M such that $L^{\Delta} = M^{\Delta}$.*

Proof. Assume a VPA $\mathcal{M}$ for L that is determinized and has the state set Q. When reading a call letter it stores it to the stack together with the present state. Storing the state is what makes $\mathcal{M}$ not being very visibly. So, for M we enrich the alphabet and let $\Sigma_{\text{call}} \times Q$ now be the set of call letters. We can now take $\mathcal{M}$ and interpret it as a VVPA accepting a language M as desired. □

The consequence of the lemma is that the Corollaries 128, 130, and 132 can be tightened to VVPLs: If we have characterized the VVPLs in $\mathsf{FO}[\mathsf{arb}]$, the characterization for VPLs follows. Also, note that it is actually sufficient to only focus on linear languages.

A language for which there is no VCL with the same height behavior is

$$\mathcal{L}_3 = L(S \to aSbc|acSb|\epsilon).$$

We do not know whether it is in $\mathsf{FO}[\mathsf{arb}]$. This language has proven to be very stubborn. The previous lemma implies that the VVPL

$$\mathcal{L}_4 = L(S \to a_1Sbc|a_2cSb|\epsilon)$$

poses an equally hard problem. One approach could be to try to show that every VVPL in $\mathsf{FO}[\mathsf{arb}]$ has a height behavior for which there exists a VCL in $\mathsf{FO}[\mathsf{arb}]$ with the same height behavior.

It is worth investigating $\mathcal{L}_3$ closer as this language seems to capture the quintessence of the difficulty of the open problem of characterizing the VPLs in $\mathsf{FO}[\mathsf{arb}]$. This language has the two rule right hand sides $aSbc$ and $acSb$. Consider in addition the symmetrical ones aSb and $acSbc$. For these four rules we find 2^4 languages; there are four rules, which might be present in the grammar or not. The resulting languages are sometimes hard and sometimes not. Table 11.1 subsumes all combinations.

In all but one of the cases, $\mathcal{L}_3$, we know the status of the language. Let us call aSb and $acSbc$ the symmetric, and $aSbc$ and $acSb$ the asymmetric rules. One can observe: As soon as a symmetric and an asymmetric rule is mixed, the language becomes $\mathbf{TC}^0$-hard, because the reduction based on the idea of Proposition 131 becomes possible. If there is no mix between the two, the language is in $\mathsf{FO}[\mathsf{arb}]$, except for the one case in line seven resulting in $\mathcal{L}_3$.

	aSb	$acSb$	$aSbc$	$acSbc$	$\in$ FO[arb]?
1	×	×	×	×	✓
2	×	×	×	✓	✓
3	×	×	✓	×	✓
4	×	×	✓	✓	×
5	×	✓	×	×	✓
6	×	✓	×	✓	×
7	×	✓	✓	×	**?**
8	×	✓	✓	✓	×
9	✓	×	×	×	✓
10	✓	×	×	✓	✓
11	✓	×	✓	×	×
12	✓	×	✓	✓	×
13	✓	✓	×	×	×
14	✓	✓	×	✓	×
15	✓	✓	✓	×	×
16	✓	✓	✓	✓	×

Table 11.1: Summary of the 16 different languages we can get from the combination of four rules of the form $S \to \ldots$. Line 7 corresponds to $\mathcal{L}_3$ and line 13 to $\mathcal{L}_1$.

To investigate $\mathcal{L}_3$ even further, consider the position in the word that marks the *turning point*, i.e. the position of maximal height within the linear word. Actually, there could be two such positions that are consecutive, but this is not relevant at this point. For a word $w \in \mathcal{L}_3$, the turning point is in the interval $[\frac{|w|}{3}, \frac{2|w|}{3}]$ where $a^n(bc)^n$ and $(ac)^n b^n$ mark the two extreme choices. Now, we could consider a modification of $\mathcal{L}_3$ where we artificially fix the turning point. If we fix it to, say, $\frac{|w|}{3}$ the language becomes easy because it then is $a^n(bc)^n$. The same is, of course, true for the other border. On the other hand, if we fix it to $\frac{|w|}{2}$ then one can see that the language is as hard as before. We also see that it does not have SHB in this case. In another case we could fix it to be within an interval $[\frac{|w|}{3}, \frac{|w|}{3} + k]$ for some constant k. This is again in FO[arb] and indeed has SHB, or equivalently, a bounded corridor of size k. Due to [DLM07] we conjecture that a turning point restriction of $[\frac{|w|}{3}, \frac{|w|}{3} + \log(|w|)]$ is still in FO[arb] but this language is not a VPL. As one can see, there are different angles of attack for this problem. For now, figuring out the status of $\mathcal{L}_3$ is the next step. A solution should yield so much insight that it can be generalized to the big problem for VPLs in general. Trying to use concepts like communication complexity, or building on the approaches of [FSS84, Hås87] that showed the lower bound for the parity language, have not been successful yet. Whether it stays that way remains to be seen.

11.5 First-Order Definability of Nested Word Languages

In the previous section we looked at the definability of languages of well-matched words. A key step was to define a matching predicate in first-order logic, but we can also consider the case where the full matching predicate $\rightsquigarrow$ is already present in the logic. This actually is the case of nested words wherein we have the matching predicate built into the structure where it may be accessed in logic. In the nested word case there are no call and return letters, however, for simplicity, we will continue where we have left off in the previous section by sticking with well-matched words and then just add the matching predicate $\rightsquigarrow$ to the logic. This approach is not restrictive. So, the questions we address in this section are which VPLs are in $\mathsf{FO}[\mathsf{arb}, \rightsquigarrow]$, $\mathsf{FO}[\mathsf{Reg}, \rightsquigarrow]$, and $\mathsf{FO}[<, \rightsquigarrow]$, for which we give partial answers.

First, we see that quasiaperiodicity still is a necessary condition.

Proposition 150. *If a VPL L is in $\mathsf{FO}[\mathsf{arb}, \rightsquigarrow]$, then it is quasiaperiodic.*

Proof. Suppose that L is not quasiaperiodic but in $\mathsf{FO}[\mathsf{arb}, \rightsquigarrow]$. The fact that L is not quasiaperiodic is either because the horizontal or the vertical condition of the quasiaperiodicity condition is violated. It would be sufficient to only use the vertical condition, however for comprehensibility we still start with the horizontal condition.

First case: The horizontal condition is violated. In this case there exists $m \in \mathbb{N}$ such that $\eta_L(\mathrm{WM}(\hat{\Sigma}) \cap \Sigma^m)$ contains a non-trivial group. In particular we then find a cyclic group $\mathbb{Z}_p$ for some prime p. Let e be the neutral element of $\mathbb{Z}_p$ and let d be some other element. Then it holds that $\langle d \rangle = \mathbb{Z}_p$. Let w_e and w_d be words in $\mathrm{WM}(\hat{\Sigma}) \cap \Sigma^m$ for which $w_e \in \eta_L^{-1}(e)$ and $w_d \in \eta_L^{-1}(d)$.

Now let $L' = \phi(\textsc{Mod}_p)$, where $\textsc{Mod}_p = \{w \in \{0,1\}^* \mid |w|_1 \bmod p = 0\}$ and ϕ is a homomorphism with $\phi(0) = w_e$ and $\phi(1) = w_d$. Since $L \in \mathsf{FO}[\mathsf{arb}, \rightsquigarrow]$, so is L'. Yet, L' has SHB because of the bounded height profile, thus, we may replace $\rightsquigarrow$ by $\rightsquigarrow_L$, which in turn is expressible by the + predicate. So, it can be concluded that L' is in $\mathsf{FO}[\mathsf{arb}]$. However, it is immediate that $\textsc{Mod}_p$ is $\mathbf{AC}^0$-reducible to L', which means that $\textsc{Mod}_p$ is in $\mathsf{FO}[\mathsf{arb}] = \mathbf{AC}^0$ from which we know that this is not the case [FSS84, Hås87, Smo87], and thus leads to a contradiction.

In the horizontal case we reduced $\textsc{Mod}_p$ to a regular word language, which we derived from L and its horizontal monoid. In the vertical case we will do the same, but the regular language has to be constructed differently. So, suppose that the vertical quasiaperiodicity condition is violated. Again, we find a group $\mathbb{Z}_p$ but this time it is spanned by two contexts $c_e = (u_e, v_e)$ and $c_d = (u_d, v_d)$ with $|u_e| = |u_d| = m$ and $|v_e| = |v_d| = n$.

We show that we can reduce $\textsc{Mod}_p$ to L, which yields the desired contradiction. for th reduction we choose the mapping $f\colon w \mapsto \alpha\phi(w)\beta\psi(w^R)\gamma$ where w^R is the reversal of w, and $\alpha\gamma$ and β are well-matched words such that $\alpha u_e \beta v_e \gamma \in L$, but $\alpha u_d \beta v_d \gamma \notin L$. Also, ϕ and ψ are homomorphisms where ϕ maps $0 \mapsto u_e$ and $1 \mapsto u_d$, and ψ maps $0 \mapsto v_e$ and $1 \mapsto v_d$. Now, the image of f is always a well-matched word and $w \in \textsc{Mod}_p$ if and only if $f(w) \in L$. The map f is computable in $\mathsf{FO}[+]$.

□

When characterizing the VPLs definable in first-order logic, we know that WSHB is not a necessary condition if the full matching predicate $\rightsquigarrow$ is assumed. For example, the Dyck language clearly is in $\mathsf{FO}[\rightsquigarrow]$. However, without WSHB we cannot apply the technique used in the proof of Proposition 137, hence we need to require WSHB in order to obtain the upper bound. We then directly get:

Proposition 151. *Quasiaperiodic VPLs with WSHB are in* $\mathsf{FO}[+, \rightsquigarrow]$.

We suspect that it could be possible to tighten the upper bound to $\mathsf{FO}[\mathsf{Reg}, \rightsquigarrow]$, however, this would need a different proof. Also, in general we conjecture that the answer to the question of whether the following holds is true:

$$\mathsf{VPL} \cap \mathsf{FO}[\mathsf{arb}, \rightsquigarrow] \overset{?}{=} \mathsf{FO}[\mathsf{Reg}, \rightsquigarrow].$$

To tackle this open problem one has to look at the WSHB definition. We see that if a language L does not have the WSHB property then there exists an element $h \in H_L$ that is a witness, i.e. $\eta_L^{-1}(h)$ contains words z, v, x such that $z = uvwxy$ for some well-matched words uy and w. Let us call such a witness element h multi-nestable. It is those elements for which we have to find a solution. This is a difficult open problem, but we can approximate and extend the approach of the proof of Proposition 137. In Section 6.5.3 we looked into the problem of deciding which VPLs are a intersection of a regular language with well-matched words or strongly well-matched words. This can be applied here. If a VPL L is such that it is quasiaperiodic and in which for all multi-nestable elements $h \in H_L$ it holds that $\eta_L^{-1}(h)$ is the intersection of a quasiaperiodic regular word language with a strongly well-matched set $\mathrm{SWM}(G)$, then L is in $\mathsf{FO}[+, \rightsquigarrow]$:

Lemma 152. *A VPL L is in* $\mathsf{FO}[+, \rightsquigarrow]$ *if L is quasiaperiodic and for every multi-nestable element $h \in H_L$ holds that $\eta_L^{-1}(h) = R \cap \mathrm{SWM}(G)$ for some quasiaperiodic regular language R and a matching graph G.*

Proof. First, we guess maximal well-matched factors and check whether they belong to some $\eta_L^{-1}(h)$ for h being multi-nestable. These factors can be evaluated in $\mathsf{FO}[\mathsf{Reg}, \rightsquigarrow]$. We replace these evaluated factors by internal letters that yield the evaluation. Thereafter, we can apply the procedure of the proof of Proposition 137. □

We do not know whether it is decidable if a VPL is the quasiaperiodic regular restriction of a set of strongly well-matched words, however, we suspect so. A first step would be to show decidability of the general nonquasiaperiodic case.

An example for a language in $\mathsf{FO}[\mathsf{Reg}, \rightsquigarrow]$ that is not captured by the property used in Lemma 152 is the language $L(S \to aSbaSb|\epsilon)$, which is represented through the formula

$$\forall x \exists y\ x \rightsquigarrow y \vee y \rightsquigarrow x \wedge \forall x, y\ \ x \rightsquigarrow y \wedge \neg\mathrm{first}(x) \wedge \neg\mathrm{last}(y) \to Q_a(x-1) \leftrightarrow Q_a(y+1).$$

This can be regarded as a restriction of the Dyck language, which only contains words that code a binary tree. So, this example does not have WSHB and is not a regular restriction of the set of well-matched words. One could try to further extend the property in order to capture such cases also.

Now that we have considered the languages in $\mathsf{FO}[\mathsf{Reg}, \rightsquigarrow]$ it is natural to look at $\mathsf{FO}[<, \rightsquigarrow]$. Here, everything works out as expected. We replace quasiaperiodicity by aperiodicity in all cases. We call a VPL *aperiodic* if both H_L and V_L are aperiodic. The proof of Proposition 150 can be easily adapted to prove the following proposition:

Proposition 153. *If a VPL L is in* $\mathsf{FO}[<, \rightsquigarrow]$*, then it is aperiodic.*

This time we conjecture that aperiodic VPLs with WSHB are in $\mathsf{FO}[<, \rightsquigarrow]$.

Because of the open questions we encountered we cannot make a statement about decidability. Of course, we suspect that it is decidable which VPLs are definable in $\mathsf{FO}[\mathsf{arb}, \rightsquigarrow]$ but this question remains open relative to the other open questions.

11.6 First-Order Definability of Tree Languages

We looked at languages of well-matched words and languages of nested words. Now, languages of actual trees are left which, as we saw, are equivalent to the mentioned word models. We again ask the question of first-order decidability and try to approach it from an angle originating in the previous results.

If asked for first-order definability of forest languages, traditionally one considers an ancestor predicate $\prec$. So, if we want to know whether some forest language is in $\mathsf{FO}[\prec]$, we will refer to this predicate. This problem is well-known and notoriously hard. We will relate this problem to the setting of words.

Recall that, given a forest language F, by $\mathsf{wm}(F)$ we denote the corresponding VPL. Note that $\mathsf{wm}(F)$ does not use any internal letters. Both F and $\mathsf{wm}(F)$, by definition, have the same syntactic forest algebra. An immediate property is that for a forest language F to be in $\mathsf{FO}[\prec]$, the horizontal monoid has to be commutative.

Proposition 154. *If a forest language F is in $\mathsf{FO}[\prec]$, then $\mathsf{wm}(F)$ is in $\mathsf{FO}[<, \rightsquigarrow]$.*

Proof. Given is ϕ, which is an $\mathsf{FO}[\prec]$ formula for F. An $\mathsf{FO}[<, \rightsquigarrow]$ formula ϕ_{WM} for $\mathsf{wm}(F)$ can be built inductively. Because of that, we also have to deal with formulas with free variables. Also, if Σ is the alphabet for F, then $(\Sigma^{\uparrow}, \Sigma^{\downarrow}, \emptyset)$ is the alphabet for $\mathsf{wm}(F)$, where $\Sigma^{\uparrow}$ and $\Sigma^{\downarrow}$ are two copies of Σ such that if $a \in \Sigma$ then there are $a^{\uparrow} \in \Sigma^{\uparrow}$ and $a^{\downarrow} \in \Sigma^{\downarrow}$.

- If ϕ is of the form $\exists x \psi$, then $\phi_{\mathrm{WM}} = \exists x_1 \exists x_2 \;\; x_1 \rightsquigarrow x_2 \wedge \psi_{\mathrm{WM}}$. So, the constructed formula has twice as many variables as the original.
- If $\phi = \neg\psi$ then $\phi_{\mathrm{WM}} = \neg\psi_{\mathrm{WM}}$.
- If $\phi = \psi^1 \wedge \psi^2$ them $\phi_{\mathrm{WM}} = \psi^1_{\mathrm{WM}} \wedge \psi^2_{\mathrm{WM}}$.
- If $\phi = Q_a(x)$ then $\phi_{\mathrm{WM}} = Q_{a^{\uparrow}}(x_1) \wedge Q_{a^{\downarrow}}(x_2)$.
- If $\phi = x \prec y$ then $\phi_{\mathrm{WM}} = y_1 < x_1 < x_2 < y_2$.

□

For the ancestor predicate $\prec$, we can define an analogue on well-matched words. Let $\lesssim$ be a binary predicate that is defined as follows: $x \lesssim y$ if and only if there exists z with $x \rightsquigarrow z$ or $z \rightsquigarrow x$ and $x < y < z$ or $z < y < x$. The predicate $\rightsquigarrow$ can be expressed by $\lesssim$, but $<$ not by $\lesssim$.

Since $\prec$ and $\lesssim$ are equivalent, we get the following:

Proposition 155. *A forest language F is in $\mathsf{FO}[\prec]$ if and only if $\mathsf{wm}(F)$ is in $\mathsf{FO}[\lesssim]$.*

We have related the problem of deciding whether a forest language is first-order definable to the problem of deciding whether a VPL is first-order definable. This leads to the partial results we showed in the previous section. Conversely, the open problems about VPLs are now also related to first-order definability of forest languages. However, the problem we face in the case of forest languages might be not as hard since we have commutativity.

11.7 Application

The main motivation to look at first-order-definability of VPLs in such a depth was the problem of evaluation of terms over finite algebras. There are, of course, the open questions inherited from the VPLs and also it remains open how to incorporate infinite algebras.

11.7.1 Term Evaluation Over Finite Algebras

If an arbitrary finite algebra is given, terms can be evaluated in $\mathbf{NC}^1$. If we want to evaluate in $\mathbf{AC}^0$, it is necessary let go of arbitrary terms as inputs. Evaluation in $\mathbf{AC}^0$ requires the input terms to have a bounded Horton-Strahler number, which is the same as WSHB. Since we have still open questions with regard to which tree shapes that actually can be evaluated in $\mathbf{AC}^0$, we do not know about a condition that is sufficient and necessary, however, SHB is at least sufficient. Also necessary for the evaluation problem to be in $\mathbf{AC}^0$ is quasiaperiodicity.

Note that terms are ranked trees. In the framework we laid out here we also can treat unranked trees and hence evaluation of algebras with finitary operations embeds naturally.

Also note that terms are trees that are encoded as words. We chose the well-matched words, which represent an in-order representation. There exist different word representations that could lead to different results.

11.7.2 Dense Completeness

This application is not related to evaluation but rather shows insights in relations between complexity classes. This is a byproduct of Corollary 145 and Proposition 144 which we want to include here.

Dense completeness describes a strong relationship between a complexity class and a *formal language class.* By formal language class we intuitively mean a class like the regular or CFLs. We lack a formal definition of what a formal language class is, but this is not necessarily a problem as we will see.

Definition 156 ([KL12]). *A formal language class $\mathcal{F}$ is densely complete in a complexity class $\mathcal{C}$ if*

- $\mathcal{F} \subseteq \mathcal{C}$ *and*
- *for all languages $C \in \mathcal{C}$ there exists a language $F \in \mathcal{F}$ such that $C \leq_{\mathbf{AC}^0} F$ and $F \leq_{\mathbf{AC}^0} C$, where $\leq_{\mathbf{AC}^0}$ indicates an $\mathbf{AC}^0$ many-one reduction.*

Of course, dense completeness can also be defined using other variants of reducibility depending on the context. Dense completeness refines the notion of completeness. For example, the regular languages are complete in $\mathbf{NC}^1$ because the regular languages are contained in $\mathbf{NC}^1$ and there exists one regular language that is $\mathbf{NC}^1$-complete. Dense completeness requires that we not only find a language in the formal language class that is as hard as the hardest problem in the complexity class but that we find an equally hard formal language for all less hard problems. The Theorem of Ladner [Lad75, Vol90] tells us that complexity is a continuum: For two problems $A \leq_{\mathbf{AC}^0} B$, we find a problem C for which $A \leq_{\mathbf{AC}^0} C \leq_{\mathbf{AC}^0} B$, assuming that $A \notin \mathbf{AC}^0$. In the case of regular languages and $\mathbf{NC}^1$, it turned out that the regular languages fall into discrete complexity levels and hence they are not densely complete in $\mathbf{NC}^1$ [KL12]. On the other hand there exist examples for dense completeness: For example, the non-deterministic one-counter languages are densely complete in $\mathbf{NL}$. In general, all positive examples we know involve non-deterministic classes, which led to the conjecture that deterministic classes do not have densely complete formal language classes. If one showed that, for example, $\mathbf{L}$ does not have any densely complete formal language classes, then $\mathbf{NL}$ and $\mathbf{L}$ would be separated. However, as we do not have a formal definition of what a formal language class is, we cannot actually argue over *all* formal language classes. However, we may fix one definition of formal language class for each problem we want to tackle. For example, if we want to show $\mathbf{NL} \neq \mathbf{L}$ it is sufficient to find a definition that encloses the non-deterministic one-counter languages and for which we can show that no such formal language class is densely complete in $\mathbf{L}$.

The previous proof strategies for these major open problems are, of course, very optimistic. Currently we try to gather more and more positive and negative examples for dense completeness. The regular languages are not densely complete in $\mathbf{NC}^1$, but this does not mean that there is not a larger class in $\mathbf{NC}^1$ that actually is densely complete. The visibly pushdown languages, for example, are a candidate for that as they generalize the regular languages. Because of the open problems discussed before, we cannot tackle this problem yet, but we can use visibly counter languages and show that they, as we suspected, are also not densely complete in $\mathbf{NC}^1$.

Theorem 157. *The visibly counter languages are not densely complete in* $\mathbf{NC}^1$.

Proof. Assume that the VCLs are densely complete in $\mathbf{NC}^1$. Consider the case where $L \in \mathbf{NC}^1$ is a language that is not in $\mathbf{ACC}^0_2 \setminus \mathbf{AC}^0$ but not $\mathbf{ACC}^0_2$-hard. Such a language exists due to Ladner's Theorem [Lad75, Vol90]. We may use a $\mathbf{NC}^1$-complete regular language and PARITY which is $\mathbf{ACC}^0_2$-complete. Ladners Theorem now gives us a language in the middle as desired.

Now because we assumed dense completeness, there must exist a VCL V that is equivalent to L via $\mathbf{AC}^0$ many-one reductions. We get the following cases:

- $V \in \mathbf{AC}^0$. In this case, L is also in $\mathbf{AC}^0$, which is a contradiction.
- $V \notin \mathbf{AC}^0$. Note that we characterized the VCLs in $\mathbf{AC}^0$ by two properties, which both have to be true for V to be in $\mathbf{AC}^0$. If one of both is violated, the language is not in $\mathbf{AC}^0$:
  - If V does not have SHB, it is $\mathbf{TC}^0$-hard (Proposition 144), and so is L. However, by construction, L is not even $\mathbf{ACC}^0_2$-hard, which is a contradiction.
  - If V is not quasiaperiodic, V and L are hard for some $\mathbf{ACC}^0_k$. If k is even, they are also hard for $\mathbf{ACC}^0_2$, which contradicts the construction of L. If k is odd, we get $\mathbf{ACC}^0_k \subseteq \mathbf{ACC}^0_2$, which is a contradiction, also [Smo87].

□

From the previous theorem we can also derive the version for regular languages, because they are part of VCL.

Corollary 158 ([KL12])**.** *The regular languages are not densely complete in* $\mathbf{NC}^1$.

As a next step we may generalize to VPLs. Another way of generalization would be to let go of visibility. We already know that non-deterministic one-counter languages are densely complete in **NL**, but we do not know whether the deterministic version is densely complete in **L**. If we look at the previous proof, we see what we could use to achieve that result. If we could show that these deterministic counter languages that are not in $\mathbf{AC}^0$ are either $\mathbf{ACC}^0_k$-hard or $\mathbf{TC}^0$-hard, we had the desired complexity gap and could prove the result similarly. To show that, we would most certainly first want to characterize the deterministic counter languages in $\mathbf{AC}^0$.

Characterizing the deterministic counter languages in $\mathbf{AC}^0$ will need different or at least extended techniques. Meaningful examples for counter languages include $(a^nba^nc)^*$ and $(a^nba^n)^*$. Note that the first language is first-order definable whereas the second is not. An approach to show $\mathbf{AC}^0$ membership for a deterministic counter language could be to label the letters according to whether they increase or decrease the counter; the result would be a VCL. In first example language this is easily possible: All positions having the letter a become call letters if the next non-a letter is b. If it is c, the letter becomes a return letter. In the second example this does not work any more. One has to look at the entire word in order to assign the call and return letters. We think that there is some *locality property* needed for a counter language to be in $\mathbf{AC}^0$.

11.8 Conclusion

Summary

In the two previous chapters we considered evaluation in general, which led to complexities in terms of logarithmic depth. It turned out that tree-shaped problems are eligible for being addressed by our framework. For example, a general graph problem drops to logarithmic depth in complexity when tree-likeness is assumed. Now, in this final chapter we explored the lower ends of evaluation complexity. We tried to capture properties that place an evaluation problem in constant depth. This is just an iteration in the research on that topic, so we cannot give a full characterization. The research done here is limited to evaluation over finite algebras, which also helps to connect it to other research areas such as formal language theory. In particular, instead of considering evaluation over finite algebras directly, we looked at visibly pushdown languages, which even generalized the problem slightly. In doing so, we are left with the question: Which VPLs are in constant depth. This question is a straight generalization of the already solved variant of the question concerning regular languages.

The basic variant of the problem in question asks for a characterization of VPLs in $\mathbf{AC}^0$. Since $\mathbf{AC}^0$ equals $\mathsf{FO}[\mathsf{arb}]$ we shifted to logic as it offers more structure. We found out different properties that limit the complexity of the input trees: SHB and WSHB. If SHB holds, a matching predicate is definable. If WSHB does not hold, the language is $\mathbf{TC}^0$-hard. Unfortunately we were not able to close the gap between SHB and WSHB. This problem we addressed individually. It turned out to be a peculiar problem: There is a very simple language for which we do not know the status of. To solve it we probably need insights being as deep as for solving the parity problem.

The complexity of the input trees was complemented by a generalization of quasiaperiodicity to VPLs. If quasiaperiodicity and WSHB are given, as well as a matching predicate, a first-order formula if definable.

For VPLs in general we have the mentioned gap, however, if we go over to VCLs, we get a complete picture again, which strictly generalizes the results about regular languages.

To approach the general result for VPLs we also looked at cases other than $\mathsf{FO}[\mathsf{arb}]$. For instance if we just add the non-numerical matching predicate to the logic we basically are in the nested word case. We also built a relation to unranked tree languages.

Finally, we covered some applications. Of course, evaluating finite algebra in itself is one and can again be used for other problems. In the end we also considered the concept of dense completeness, which relates rather to the formal language interpretation of our research. Here, we showed that VCLs are not densely complete in $\mathbf{NC}^1$.

Contributions

Everything we presented in this chapter is new and parts have been published previously [KLL15b, KLL15a]. In [KLL15b] the result about first-order-definability of VCLs was obtained. Note that this result now is a mere corollary within a greater framework. The proof in the paper is rather combinatorial while the approach in the present work is more algebraic.

Also we contributed the dense completeness result for VCLs.

Sources and Related Work

In addition to the works introducing logic and automata models the most important pillar this chapter stands on are the contributions about the relationship of regular word languages and constant depth circuits. Those are the ones that characterize the regular languages in $\mathbf{AC}^0$ by quasiaperiodicity. Consulate the conclusion sections of the respective chapters for source details.

We published the results about VCLs in [KLL15b] and the dense completeness result was part of [KLL15a]. The former recieved more attention resulting in a follow-up paper [HKLL15] which related VCLs to classes within $\mathbf{NC}^1$ other than $\mathbf{AC}^0$. Our dense completeness result on VCLs can be found in [KLL15a] while dense completeness itself has been introduced in [KL12].

In [EJT12] Elberfeld et al. investigate Courcelle's Theorem closer under the assumption that input is an already decomposed tree. In general, checking whether the decomposition satisfies the MSO formula is in $\mathbf{NC}^1$, but they also considered in which cases this can be done in $\mathbf{AC}^0$. They show a property that is sufficient for placing the problem in $\mathbf{AC}^0$. The property basically says that the graph has to be star-like, which seems to be a special case of the SHB property.

Further Research

The goal is to obtain a complete description of evaluation capabilities in low complexity. So, as a next step the mentioned open problem should be addressed. This then solves the case of finite algebras. After, one can go after some infinite algebras. On the other hand, one can look at other classes than $\mathbf{AC}^0$ by taking up the investigations we initiated in [HKLL15]

We provided properties that are decidable, however, the exact complexity is not yet determined. The complexity of computing the Horton-Strahler number of a tree is a problem for which it would be also interesting to investigate upper bounds. One could try to place it in $\mathbf{NC}^1$, maybe by using our evaluation framework.

Another aspect that deserves attention is the relationship of the properties developed here to, say, the work of Elberfeld et al. [EJT12] and path width or tree depth.

One more research question would be, whether there is a connection between, say, the SHB property and properties of syntactic algebras. This is of similar flavor as the problem we already mentioned in the context of cost register automata where we asked whether there is a connection between copylessness and the transformation monoid. Also the reverse question is relevant: Suppose the horizontal or the vertical monoid of the syntactic algebra is, say, aperiodic, what does this mean for the height behavior of the language?

Chapter 12

Conclusion

In this thesis we provided material for the four central topics:

- A general term evaluation framework
- A template for applying this framework as well as a set of concrete applications
- Analysis of term evaluation capabilities of constant depth classes using VPLs.
- Automata theory related to term evaluation, in particular VPAs, CRAs and the combination of both.

In summary, the most notable contributions are the following:

- **Developing the machinery to formally express all results within the same setting.** This includes notions such as many-sorted families of algebras, generalized homomorphisms, generalized and many-sorted circuits.
- **Extend algebras and algebras for well-matched word languages.** Extend algebras complement forest algebras well and both are suited for capturing VPLs algebraically.
- **Complementing the already known forest algebra with extend algebras.** We also showed how these algebras that are tailored to forest languages can be used for languages for well-matched and nested words.
- **Marrying VPAs and CRAs.** We showed a model that incorporated a visible stack and cost registers. This model is still tame while exhibiting great and meaningful expressibility. For instance, it is sufficiently powerful to perform term evaluation.

- **Generalizing copylessness.** CRAs and CVPAs can be restricted resulting in copyless machines. This prevents unwanted generality by excluding functions having outputs that need exponential size to represent. However, we discovered that copylessness is unnecessary restrictive. Polynomial boundedness is a probate alternative. Copylessness can be embedded as it coincides with linear boundedness.
- **Providing a general term evaluation algorithm.** It works with any algebra $\mathcal{A}$ and leads to a complexity of $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$.
- **Providing a template for using the term evaluation algorithm.** It may be used to obtain upper bounds for a wide variety of problems. We reproved many known results within this uniform setting.
- **Showing new results using the template as well as reproving old ones.** One major block of results consists of CVPAs, in particular polynomially bounded ones. Others concern bounded width circuits and **NP** problems.
- **Analyzing evaluation in constant depth by considering first-order definability of VPLs.** We greatly generalized the result that characterizes the VCLs in $\mathbf{AC}^0$ and provided algebraic proofs. We showed necessary and sufficient conditions for VPLs being in $\mathbf{AC}^0$, however, not a single condition that is both. Therein we discovered the very interesting language $L(S \to acSb \mid S \to aSbc \mid \epsilon)$ for which we do not know whether it is in $\mathbf{AC}^0$. This language could be a starting point for research that lets us understand $\mathbf{AC}^0$ better.

We can draw a few conclusions. First, capturing VPLs algebraically seems promising. Extend and forest algebras proved to be very useful. They complement each other as both are useful in different situations.

Secondly: Our attempt to unify and simplify the large set of problems using term evaluation was fruitful. When trying to obtain upper bounds for a problem that is in some way tree-structured, it is a sensible approach to first consult the template developed here. It might spare one from designing an involved proof. Third: From a more practical perspective, a constant goal in automata theory is to come up with models that have a good tradeoff between expressibility on the one hand and complexity and closure properties on the other hand. CVPAs combine two models with a good tradeoff into a new one. Polynomial boundedness extends this good tradeoff whereas now copylessness has been the best we know. Finally: Characterizing the VPLs in $\mathbf{AC}^0$ is a very hard problem. Solving it would involve learning significant lessons about $\mathbf{AC}^0$.

In the future, research can go into different directions. Exploring more problems that can be tackled using our template would be an obvious goal. There should be many more problems that could be brought down from polynomial time into parallel complexity. Also, analyzing and applying CVPAs seems promising. This model has the potential to be relevant whenever either CRAs or VPAs already are. Polynomial boundedness should be researched further as well. Lastly, figuring out the complexity of the language $L(S \rightarrow acSb \mid S \rightarrow aSbc \mid \epsilon)$ would be insightful. A solution should give us deeper insights in to $\mathbf{AC}^0$. We conjecture that a solution can be generalized to the whole of all VPLs. However, it should be stressed that this problem is a very hard one. For instance it would most probably provide a true alternative proof for $\mathbf{AC}^0 \neq \mathbf{TC}^0$.

Bibliography

[AAD00] Manindra Agrawal, Eric Allender, and Samir Datta. On $\mathbf{TC}^0$, $\mathbf{AC}^0$, and Arithmetic Circuits. *J. Comput. Syst. Sci.*, 60(2):395–421, 2000.

[ADD$^+$11] Rajeev Alur, Loris D'Antoni, Jyotirmoy V. Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular Functions, Cost Register Automata, and Generalized Min-Cost Problems. *CoRR*, abs/1111.0670, 2011.

[ADD$^+$13] Rajeev Alur, Loris D'Antoni, Jyotirmoy V. Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular Functions and Cost Register Automata. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 13–22. IEEE Computer Society, 2013.

[AKM17] Eric Allender, Andreas Krebs, and Pierre McKenzie. Better Complexity Bounds for Cost Register Machines. *Electronic Colloquium on Computational Complexity (ECCC)*, 24:72, 2017.

[AKMV05] Rajeev Alur, Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Congruences for Visibly Pushdown Languages. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, volume 3580 of *Lecture Notes in Computer Science*, pages 1102–1114. Springer, 2005.

[All04] Eric Allender. Arithmetic circuits and counting complexity classes. 2004.

[Alm94] J. Almeida. *Finite Semigroups and Universal Algebra.* Series in algebra. World Scientific, 1994.

[AM04] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 202–211. ACM, 2004.

[AM09] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3):16:1–16:43, 2009.

[AM15] Eric Allender and Ian Mertz. Complexity of Regular Functions. In Adrian Horia Dediu, Enrico Formenti, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings*, volume 8977 of *Lecture Notes in Computer Science*, pages 449–460. Springer, 2015.

[BCGK17] Célia Borlido, Silke Czarnetzki, Mai Gehrke, and Andreas Krebs. Stone Duality and the Substitution Principle. In Valentin Goranko and Mads Dam, editors, *26th EACSL Annual Conference on Computer Science Logic, CSL 2017, August 20-24, 2017, Stockholm, Sweden*, volume 82 of *LIPIcs*, pages 13:1–13:20. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[BCGR92] Samuel R. Buss, Stephen A. Cook, A. Gupta, and V. Ramachandran. An Optimal Parallel Algorithm for Formula Evaluation. *SIAM J. Comput.*, 21(4):755–780, 1992.

[BCK$^+$14] Achim Blumensath, Thomas Colcombet, Denis Kuperberg, Pawel Parys, and Michael Vanden Boom. Two-way cost automata and cost logics over infinite trees. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 16:1–16:9. ACM, 2014.

[BCST92] David A. Mix Barrington, Kevin J. Compton, Howard Straubing, and Denis Thérien. Regular Languages in $\mathbf{NC}^1$. *J. Comput. Syst. Sci.*, 44(3):478–499, 1992.

[BDG15] Nikhil Balaji, Samir Datta, and Venkatesh Ganesan. Counting Euler Tours in Undirected Bounded Treewidth Graphs. In Prahladh Harsha and G. Ramalingam, editors, *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015, December 16-18, 2015, Bangalore, India*, volume 45 of *LIPIcs*, pages 246–260. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[BKR11] Christoph Behle, Andreas Krebs, and Stephanie Reifferscheid. Typed Monoids - An Eilenberg-Like Theorem for Non Regular Languages. In Franz Winkler, editor, *Algebraic Informatics - 4th International Conference, CAI 2011, Linz, Austria, June 21-24, 2011. Proceedings*, volume 6742 of *Lecture Notes in Computer Science*, pages 97–114. Springer, 2011.

[BL06] Christoph Behle and Klaus-Jörn Lange. $\mathsf{FO}[<]$-Uniformity. In *21st Annual IEEE Conference on Computational Complexity (CCC 2006), 16-20 July 2006, Prague, Czech Republic*, pages 183–189. IEEE Computer Society, 2006.

[BLS06] Vince Bárány, Christof Löding, and Olivier Serre. Regularity Problems for Visibly Pushdown Languages. In Bruno Durand and Wolfgang Thomas, editors, *STACS 2006, 23rd Annual Symposium on Theoretical Aspects of Computer Science, Marseille, France, February 23-25, 2006, Proceedings*, volume 3884 of *Lecture Notes in Computer Science*, pages 420–431. Springer, 2006.

[Bre74] Richard P. Brent. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM*, 21(2):201–206, 1974.

[BSW12] Mikołaj Bojańczyk, Howard Straubing, and Igor Walukiewicz. Wreath Products of Forest Algebras, with Applications to Tree Logics. *Logical Methods in Computer Science*, 8(3), 2012.

[Bus87] Samuel R. Buss. The Boolean Formula Value Problem Is in ALOGTIME. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 123–131, 1987.

[Bus93] Samuel R. Buss. Algorithms for boolean formula evaluation and for tree contraction. In *Arithmetic, Proof Theory and Computational Complexity*, pages 96–115. Oxford University Press, 1993.

[BW08] Mikołaj Bojańczyk and Igor Walukiewicz. Forest algebras. In Jörg Flum, Erich Grädel, and Thomas Wilke, editors, *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas].*, volume 2 of *Texts in Logic and Games*, pages 107–132. Amsterdam University Press, 2008.

[Bü60] J. Richard Büchi. Weak Second-Order Arithmetic and Finite Automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960.

[CDG$^+$07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[CF16] Thomas Colcombet and Nathanaël Fijalkow. The Bridge Between Regular Cost Functions and Omega-Regular Languages. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 126:1–126:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[Cho95] Richard Chorley. Horton, R.E. 1945: Erosional development of streams and their drainage basins: hydrophysical approach to quantitative morphology. Bulletin of the Geological Society of America 56, 2 75-3 70. *Progress in Physical Geography*, 19(4):533–554, 1995.

[CK16] Silke Czarnetzki and Andreas Krebs. Using Duality in Circuit Complexity. In Adrian-Horia Dediu, Jan Janousek, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 10th International Conference, LATA 2016, Prague, Czech Republic, March 14-18, 2016, Proceedings*, volume 9618 of *Lecture Notes in Computer Science*, pages 283–294. Springer, 2016.

[CKL15] Michaël Cadilhac, Andreas Krebs, and Nutan Limaye. Value Automata with Filters. *CoRR*, abs/1510.02393, 2015.

[CKLP15] Michaël Cadilhac, Andreas Krebs, Michael Ludwig, and Charles Paperman. A Circuit Complexity Approach to Transductions. In Giuseppe F. Italiano, Giovanni Pighizzini, and Donald Sannella, editors, *Mathematical Foundations of Computer Science 2015 - 40th International Symposium, MFCS 2015, Milan, Italy, August 24-28, 2015, Proceedings, Part I*, volume 9234 of *Lecture Notes in Computer Science*, pages 141–153. Springer, 2015.

[CL10] Thomas Colcombet and Christof Löding. Regular Cost Functions over Finite Trees. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*, pages 70–79. IEEE Computer Society, 2010.

[CMTV98] Hervé Caussinus, Pierre McKenzie, Denis Thérien, and Heribert Vollmer. Nondeterministic $\mathbf{NC}^1$ Computation. *J. Comput. Syst. Sci.*, 57(2):200–212, 1998.

[CO00] Bruno Courcelle and Stephan Olariu. Upper bounds to the clique width of graphs. *Discrete Applied Mathematics*, 101(1-3):77–114, 2000.

[Col09] Thomas Colcombet. The Theory of Stabilisation Monoids and Regular Cost Functions. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikoletseas, and Wolfgang Thomas, editors, *Automata, Languages and Programming, 36th Internatilonal Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part II*, volume 5556, pages 139–150. Springer, 2009.

[Col13] Thomas Colcombet. Regular Cost Functions, Part I: Logic and Algebra over Words. *Logical Methods in Computer Science*, 9(3), 2013.

[Col17] Thomas Colcombet. Logic and regular cost functions. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–4. IEEE Computer Society, 2017.

[Coo71] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[Coo85] Stephen A. Cook. A Taxonomy of Problems with Fast Parallel Algorithms. *Information and Control*, 64(1-3):2–21, 1985.

[Cou90] Bruno Courcelle. The Monadic Second-Order Logic of Graphs. I. Recognizable Sets of Finite Graphs. *Inf. Comput.*, 85(1):12–75, 1990.

[DA14] Loris D'Antoni and Rajeev Alur. Symbolic Visibly Pushdown Automata. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 209–225. Springer, 2014.

[DFKL14] Olga Dorzweiler, Thomas Flamm, Andreas Krebs, and Michael Ludwig. Positive and Negative Proofs for Circuits and Branching Programs. In *Descriptional Complexity of Formal Systems - 16th International Workshop, DCFS 2014, Turku, Finland, August 5-8, 2014. Proceedings*, pages 270–281, 2014.

[DFKL16] Olga Dorzweiler, Thomas Flamm, Andreas Krebs, and Michael Ludwig. Positive and negative proofs for circuits and branching programs. *Theor. Comput. Sci.*, 610:24–36, 2016.

[Die12] Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

[DLM07] Arnaud Durand, Clemens Lautemann, and Malika More. A simple proof of the polylog counting ability of first-order logic: guest column. *SIGACT News*, 38(4):40–45, 2007.

[Don70] John Doner. Tree Acceptors and Some of Their Applications. *J. Comput. Syst. Sci.*, 4(5):406–451, 1970.

[Dym88] Patrick W. Dymond. Input-Driven Languages are in log n Depth. *Inf. Process. Lett.*, 26(5):247–250, 1988.

[EF95] Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite model theory*. Perspectives in Mathematical Logic. Springer, 1995.

[EJT10] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Logspace Versions of the Theorems of Bodlaender and Courcelle. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 143–152. IEEE Computer Society, 2010.

[EJT12] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Algorithmic Meta Theorems for Circuit Classes of Constant and Logarithmic Depth. In Christoph Dürr and Thomas Wilke, editors, *29th International Symposium on Theoretical Aspects of Computer Science, STACS 2012, February 29th - March 3rd, 2012, Paris, France*, volume 14 of *LIPIcs*, pages 66–77. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations und Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[FSS84] Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, Circuits, and the Polynomial-Time Hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984.

[GKP14] Mai Gehrke, Andreas Krebs, and Jean-Éric Pin. From Ultrafilters on Words to the Expressive Power of a Fragment of Logic. In *Descriptional Complexity of Formal Systems - 16th International Workshop, DCFS 2014, Turku, Finland, August 5-8, 2014. Proceedings*, pages 138–149, 2014.

[GL17] Moses Ganardi and Markus Lohrey. A universal tree balancing theorem. *CoRR*, abs/1704.08705, 2017.

[GM18] Paul Gastin and Benjamin Monmege. A unifying survey on weighted logics and weighted automata - Core weighted logic: minimal and versatile specification of quantitative properties. *Soft Comput.*, 22(4):1047–1065, 2018.

[Gup85] A. Gupta. A fast parallel algorithm for recognition of parenthesis languages. Master's thesis, 1985.

[Hal76] Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8(1):171–186, 1976.

[Hås87] Johan Håstad. *Computational Limitations of Small-depth Circuits.* MIT Press, Cambridge, MA, USA, 1987.

[HKLL15] Michael Hahn, Andreas Krebs, Klaus-Jörn Lange, and Michael Ludwig. Visibly Counter Languages and the Structure of $\mathbf{NC}^1$. In Giuseppe F. Italiano, Giovanni Pighizzini, and Donald Sannella, editors, *Mathematical Foundations of Computer Science 2015 - 40th International Symposium, MFCS 2015, Milan, Italy, August 24-28, 2015, Proceedings, Part II*, volume 9235 of *Lecture Notes in Computer Science*, pages 384–394. Springer, 2015.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[JS14] Maurice J. Jansen and Jayalal Sarma. Balancing Bounded Treewidth Circuits. *Theory Comput. Syst.*, 54(2):318–336, 2014.

[Kar72] Richard M. Karp. Reducibility Among Combinatorial Problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[KL12] Andreas Krebs and Klaus-Jörn Lange. Dense Completeness. In Hsu-Chun Yen and Oscar H. Ibarra, editors, *Developments in Language Theory - 16th International Conference, DLT 2012, Taipei, Taiwan, August 14-17, 2012. Proceedings*, volume 7410 of *Lecture Notes in Computer Science*, pages 178–189. Springer, 2012.

[KLL14] Andreas Krebs, Klaus-Jörn Lange, and Michael Ludwig. Visibly Counter Languages and Constant Depth Circuits. *Electronic Colloquium on Computational Complexity (ECCC)*, 21:177, 2014.

[KLL15a] Andreas Krebs, Klaus-Jörn Lange, and Michael Ludwig. On Distinguishing $\mathbf{NC}^1$ and $\mathbf{NL}$. In Igor Potapov, editor, *Developments in Language*

Theory - 19th International Conference, DLT 2015, Liverpool, UK, July 27-30, 2015, Proceedings., volume 9168 of *Lecture Notes in Computer Science*, pages 340–351. Springer, 2015.

[KLL15b] Andreas Krebs, Klaus-Jörn Lange, and Michael Ludwig. Visibly Counter Languages and Constant Depth Circuits. In Ernst W. Mayr and Nicolas Ollinger, editors, *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany*, volume 30 of *LIPIcs*, pages 594–607. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[KLL16] Andreas Krebs, Nutan Limaye, and Michael Ludwig. Cost Register Automata for Nested Words. In Thang N. Dinh and My T. Thai, editors, *Computing and Combinatorics - 22nd International Conference, COCOON 2016, Ho Chi Minh City, Vietnam, August 2-4, 2016, Proceedings*, volume 9797 of *Lecture Notes in Computer Science*, pages 587–598. Springer, 2016.

[KLL17a] Andreas Krebs, Nutan Limaye, and Michael Ludwig. A Unified Method for Placing Problems in Polylogarithmic Depth. In Satya V. Lokam and R. Ramanujam, editors, *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017, December 11-15, 2017, Kanpur, India*, volume 93 of *LIPIcs*, pages 36:36–36:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[KLL17b] Andreas Krebs, Nutan Limaye, and Michael Ludwig. A Unified Method for Placing Problems in Polylogarithmic Depth. *Electronic Colloquium on Computational Complexity (ECCC)*, 24:19, 2017.

[KLM10] Andreas Krebs, Nutan Limaye, and Meena Mahajan. Counting Paths in VPA Is Complete for $\#\mathbf{NC}^1$. In My T. Thai and Sartaj Sahni, editors, *Computing and Combinatorics, 16th Annual International Conference, COCOON 2010, Nha Trang, Vietnam, July 19-21, 2010. Proceedings*, volume 6196 of *Lecture Notes in Computer Science*, pages 44–53. Springer, 2010.

[KLM12] Andreas Krebs, Nutan Limaye, and Meena Mahajan. Counting Paths in VPA Is Complete for $\#\mathbf{NC}^1$. *Algorithmica*, 64(2):279–294, 2012.

[Kop16] Eryk Kopczynski. Invisible Pushdown Languages. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 867–872. ACM, 2016.

[KS15] Andreas Krebs and Howard Straubing. EF+EX Forest Algebras. In Andreas Maletti, editor, *Algebraic Informatics - 6th International Conference, CAI 2015, Stuttgart, Germany, September 1-4, 2015. Proceedings*, volume 9270 of *Lecture Notes in Computer Science*, pages 128–139. Springer, 2015.

[Lad75] Richard E. Ladner. On the Structure of Polynomial Time Reducibility. *J. ACM*, 22(1):155–171, 1975.

[LM98] Klaus-Jörn Lange and Pierre McKenzie. On the Complexity of Free Monoid Morphisms. In Kyung-Yong Chwa and Oscar H. Ibarra, editors, *Algorithms and Computation, 9th International Symposium, ISAAC '98, Taejon, Korea, December 14-16, 1998, Proceedings*, volume 1533 of *Lecture Notes in Computer Science*, pages 247–256. Springer, 1998.

[LMM09] Nutan Limaye, Meena Mahajan, and Antoine Meyer. On the Complexity of Membership and Counting in Height-Deterministic Pushdown Automata, journal=Journal of Automata, Languages and Combinatorics. 14(3/4):211–235, 2009.

[Loh01] Markus Lohrey. On the Parallel Complexity of Tree Automata. In Aart Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*, volume 2051 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2001.

[LST94] Clemens Lautemann, Thomas Schwentick, and Denis Thérien. Logics for context-free languages. In Leszek Pacholski and Jerzy Tiuryn, editors, *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers*, volume 933 of *Lecture Notes in Computer Science*, pages 205–216. Springer, 1994.

[Lyn77] Nancy A. Lynch. Log Space Recognition and Translation of Parenthesis Languages. *J. ACM*, 24(4):583–590, 1977.

[Meh80] Kurt Mehlhorn. Pebbling Moutain Ranges and its Application of DCFL-Recognition. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings*, volume 85 of *Lecture Notes in Computer Science*, pages 422–435. Springer, 1980.

[MP71] Robert McNaughton and Seymour A. Papert. *Counter-Free Automata (M.I.T. Research Monograph No. 65).* The MIT Press, 1971.

[MR15] Filip Mazowiecki and Cristian Riveros. Maximal Partition Logic: Towards a Logical Characterization of Copyless Cost Register Automata. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany*, volume 41 of *LIPIcs*, pages 144–159. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[MR16] Filip Mazowiecki and Cristian Riveros. Copyless Cost-Register Automata: Structure, Expressiveness, and Closure Properties. In Nicolas Ollinger and Heribert Vollmer, editors, *33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, February 17-20, 2016, Orléans, France*, volume 47 of *LIPIcs*, pages 53:1–53:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[Neb00] Markus E. Nebel. On the Horton-Strahler number for combinatorial tries. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications*, 34(4):279–296, 2000.

[NS07] Dirk Nowotka and Jirí Srba. Height-Deterministic Pushdown Automata. In Ludek Kucera and Antonín Kucera, editors, *Mathematical Foundations of Computer Science 2007, 32nd International Symposium, MFCS 2007, Ceský Krumlov, Czech Republic, August 26-31, 2007, Proceedings*, volume 4708 of *Lecture Notes in Computer Science*, pages 125–134. Springer, 2007.

[OS06] Sang-il Oum and Paul D. Seymour. Approximating clique-width and branch-width. *J. Comb. Theory, Ser. B*, 96(4):514–528, 2006.

[Par66] Rohit Parikh. On Context-Free Languages. *J. ACM*, 13(4):570–581, 1966.

[Ram86] V. Ramachandran. Restructuring formula trees. Unpublished manuscript, 1986.

[Ruz81] Walter L. Ruzzo. On Uniform Circuit Complexity. *J. Comput. Syst. Sci.*, 22(3):365–383, 1981.

[Sch65] Marcel Paul Schützenberger. On Finite Monoids Having Only Trivial Subgroups. *Information and Control*, 8(2):190–194, 1965.

[Smo87] Roman Smolensky. Algebraic Methods in the Theory of Lower Bounds for Boolean Circuit Complexity. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 77–82, 1987.

[Spi71] P.M. Spira. On time hardware complexity tradeoffs for Boolean functions. *Proceedings of the Fourth Hawaii International Symposium on System Sciences*, pages 525–527, 1971.

[Str52] A. N. Strahler. Hypsometric Area-Altitude Analysis of Erosional Topography. *Geological Society of America Bulletin*, 63:1117, 1952.

[Str57] A. N. Strahler. Quantitative analysis of watershed geomorphology. *American Geophysical Union Transactions*, 38(6):912–920, 1957.

[Str94] Howard Straubing. *Finite Automata, Formal Logic, and Circuit Complexity.* Birkhäuser, Boston, 1994.

[Str13] Howard Straubing. New applications of the wreath product of forest algebras. *RAIRO - Theor. Inf. and Applic.*, 47(3):261–291, 2013.

[Tho97] Wolfgang Thomas. *Languages, Automata, and Logic*, pages 389–455. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[Til87] Bret Tilson. Categories as algebra: An essential ingredient in the theory of monoids. 48:83–198, 09 1987.

[TW68] James W. Thatcher and Jesse B. Wright. Generalized Finite Automata Theory with an Application to a Decision Problem of Second-Order Logic. *Mathematical Systems Theory*, 2(1):57–81, 1968.

[Vol90] Heribert Vollmer. The Gap-Language-Technique Revisited. In Egon Börger, Hans Kleine Büning, Michael M. Richter, and Wolfgang Schönfeld, editors, *Computer Science Logic, 4th Workshop, CSL '90, Heidelberg, Germany, October 1-5, 1990, Proceedings*, volume 533 of *Lecture Notes in Computer Science*, pages 389–399. Springer, 1990.

[Vol99] Heribert Vollmer. *Introduction to Circuit Complexity - A Uniform Approach.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 1999.

[Wan94] Egon Wanke. k-NLC Graphs and Polynomial Algorithms. *Discrete Applied Mathematics*, 54(2-3):251–266, 1994.

[Wir90] Martin Wirsing. Algebraic specification. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 675–788. 1990.

List of Figures

List of Symbols and Notation

This list summarizes the notation that is defined throughout this work, as well as letters that are commonly used in a certain role.

Numbers and sets

$\mathbb{N}$	the natural numbers
$\mathcal{N}$	the algebra of natural numbers: $(\mathbb{N}; +, \times, 0, 1)$
$\mathbb{Z}$	the integers
$\mathcal{Z}$	the algebra of integers: $(\mathbb{Z}; +, \times, 0, 1)$
$\mathbb{B}$	the set of truth values: $\{\bot, \top\}$
$\mathcal{B}$	the two-element Boolean algebra: $(\mathbb{B}; \wedge, \vee, \bot, \top)$
m, n, i, j	natural numbers
$[m, n]$	interval of natural numbers from m to n
$[n]$	$= [1, n]$
2^X	power set of X; equivalent to $\mathcal{P}(X)$
X^n	set of sequences/words/tuples of elements of X of length n
X^*	set of sequences/words/tuples of elements of X of finite length
$X^{m\times n}$	set of $m \times n$ matrices over X
$\sim$	relation
$X/{\sim}$	quotient of X
$[x]_\sim$	equivalence class induced by $\sim$ that contains x
$f\colon x \mapsto y$	function f maps x to y; equivalent to $f(x) = y$ and $(x, y) \in f$

Letters and words

Σ	alphabet
w	word
L	language
ϵ	the empty word
$\hat{\Sigma}$	visible alphabet consisting of call letters Σ_{call}, return letters Σ_{ret}, and internal letters Σ_{ret}
a, b, c	letters of Σ_{call}, Σ_{ret}, and Σ_{int}
Δ	height function of the form $\hat{\Sigma}^* \to \mathbb{Z}$ for well-matched words
$\rightsquigarrow$	binary matching relation in nested and well-matched words
$([n]; <, (Q_a)_{a\in\Sigma})$	word structure
$([n]; <, (Q_a)_{a\in\Sigma}, \rightsquigarrow)$	nested word structure
$w \in \Sigma^*$	word as a sequence
w_1w_2	concatenation of words
w_i	letter in position i; equivalent to $w(i)$
$\|w\|$	length of w
$\|w\|_X$	number of positions having a letter in X
$\|w\|_a$	$= \|w\|_{\{a\}}$
nw	maps well-matched words and forests onto the corresponding nested word
forest	maps nested and well-matched words onto the corresponding forest
wm	maps nested words and forests onto the corresponding well-matched word
$\mathrm{WM}(\hat{\Sigma})$	set of well-matched words over $\hat{\Sigma}$

Automata and circuits

$\mathcal{A}$-$\mathbf{NC}^1$	complexity class defined by $\mathbf{NC}^1$ circuits that are augmented by algebra $\mathcal{A}$
mp_X	multiplexer over set X
$\mathrm{eval}_{\mathcal{A}}(C)$	evaluation function realized by circuit C over algebra $\mathcal{A}$
$L(X)$	language accepted by automaton or circuit X
R^X_σ	$= \big((\mathbb{T}^X(\sigma))^X; \odot\big)$, which is the register algebra of registers X and signature σ

$\nu_0\colon X \to \mathbb{D}$	initial register valuation in a CRA/CVPA
$\rho\colon Q \times \Sigma \to R_\sigma^X$	register update function in a CRA
ρ_{call}	register update function for call letters in a CVPA of the form $Q \times \Sigma_{\text{call}} \to (\mathbb{T}^X(\sigma))^X$
ρ_{ret}	register update function for return letters in a CVPA of the form $Q \times \Sigma_{\text{ret}} \times \Gamma \to (\mathbb{T}^{X \cup X_{\text{match}}}(\sigma))^X$
ρ_{int}	register update function for internal letters in a CVPA of the form $Q \times \Sigma_{\text{int}} \to (\mathbb{T}^X(\sigma))^X$
$\mu\colon Q \to \mathbb{T}^X(\sigma)$	final cost function in a CRA/CVPA
$F_{\mathcal{A}}(\mathcal{M})\colon \Sigma^* \to \mathbb{D}$	function realized by a CRA/CVPA

Structures and graphs

$(\mathbb{D}; R)$	structure with domain $\mathbb{D}$ and family of relations R
σ	single- or many-sorted signature
S	sort
$\sigma(S)$	signature of a structure S
$G = (V; E)$	graph
$(V; E, (Q_a)_{a \in \Sigma})$	labeled graph; equivalent to $(V; E, l)$ for a labeling function $l\colon V \to \Sigma$
V	set of vertices
$V(G)$	vertex set of graph G
E	set of edges
$E(G)$	edge set of graph G
v	vertex
e	edge
$d(v)$	degree of vertex v in a graph
t	tree
f	forest
(Σ, r)	ranked alphabet where $r\colon \Sigma \to \mathbb{N}$
$t_1 + \ldots + t_n$	forest containing of n trees; $+$ may be commutative
$a(t_1 + \ldots + t_n)$	unranked labeled tree with a root labeled a; a acts as a unary function
$a(t_1, \ldots, t_k)$	ranked labeled tree with a root labeled a; a acts as a k-ary function

Algebras and homomorphisms

$\mathcal{A} = (\mathbb{D}; O)$	single- or many-sorted algebra with domain $\mathbb{D}$ and family of operations O
$\mathcal{F}(\mathcal{A})$	$= (\mathbb{D}, \widetilde{\mathbb{D}}; B, Z, \widetilde{B}, \circ, \odot, \mathrm{id})$; functional algebra of $\mathcal{A}$, where $\widetilde{\mathbb{D}} \subseteq \mathbb{D}^{\mathbb{D}}$
id	identity map
$\circ$	composition of functions
$\odot$	substitution in functions
$\sigma(\mathcal{A})$	signature of an algebra $\mathcal{A}$
$([S]^* \times [S])^k$	set of signatures of algebras of S sorts and k operations
$\circledast$, $\otimes$, $\odot$, $\oplus$	operations
$\mathsf{In}_\sigma(i)$	input signature of the i'th operation
$\mathsf{In}_\sigma(i, j)$	sort of the j'th input of the i'th operation
$\mathsf{Ar}_\sigma(i)$	arity of the i'th operation
$\mathsf{Out}_\sigma(i)$	sort of the image of the i'th operation
$\sigma(i)$	signature of the i'th operation
$\sigma(\circledast)$	$= \sigma(i)$ if $\circledast$ is the i'th operation
$(\mathcal{A}_n)_{n\in\mathbb{N}}$	family of algebras
ϕ, ψ	homomorphisms
$\ker(\phi)$	kernel of homomorphism ϕ
$\mathcal{A} \prec \mathcal{B}$	algebra $\mathcal{A}$ divides algebra $\mathcal{B}$
$\pi_\sim$	natural homomorphism $\mathcal{A} \to \mathcal{A}/{\sim}$
$\mathcal{A}/{\sim}$	$= (\mathbb{D}/{\sim}; O/{\sim})$ quotient of $\mathcal{A}$ under congruence $\sim$
$\sim_L$	syntactic congruence
η_L	$= \pi_{\sim_L}$, which is the syntactic morphism of L
$\mathrm{Synt}(L)$	syntactic algebra of L

Terms and evaluation

t	term
$\xi \colon [n] \to [S]$	variable signature, which assigns each of the n variables a sort
$\mathbb{T}^\xi_s(\sigma)$	set of terms having variable signature ξ that evaluate to a term of sort s
$\mathbb{T}^*_s(\sigma)$	set of terms having arbitrary consistent variables that evaluate to sort s

$\mathbb{C}^s(\sigma)$	set of contexts that evaluate to sort s
ν	variable valuation of the form $[n] \to \mathbb{D}$
$\mathrm{eval}^{\nu}_{\mathcal{A}} \colon \mathbb{T}^{\xi}(\sigma) \to \mathbb{D}$	evaluation function using variable valuation ν
$\mathrm{eval}_{\mathcal{A}}$	$= \mathrm{eval}^{\emptyset}_{\mathcal{A}}$
$i <_T j$	in PNF term T, position i is a child of position j
$i \lhd_T j$	in PNF term T, $[i, j]$ is a term

Forest and extend algebras

F	forest
$\mathsf{EA}(\Sigma)$	$= (H(\Sigma); +, (\triangle_a)_{a\in\Sigma}, 0)$
$\mathsf{FA}(\Sigma)$	$= (H(\Sigma), V(\Sigma); +, +', +'', \cdot, \cdot', (\triangle_a)_{a\in\Sigma}, 0, 1)$
V	vertical monoid
H	horizontal monoid
h	element of horizontal monoid
v	element of vertical monoid
$+$	horizontal concatenation of forests
$\triangle_a$	unary extend operation in extend algebra, or constant in forest algebra respectively
$\mathrm{Synt}(F)$	syntactic extend/forest algebra
$H(\Sigma)$	horizontal monoid of the free extend/forest algebra over Σ; contains all forests
$V(\Sigma)$	horizontal monoid of the free forest algebra over Σ; contains all contexts
H_F	horizontal monoid of the syntactic extend/forest algebra of F
V_F	vertical monoid of the syntactic forest algebra of F

Index