FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN
Lehrstuhl für Informatik VII

Algebraic Systems of Fixpoint Equations over Semirings: Theory and Applications

Maximilian Schlund

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN
Lehrstuhl für Informatik VII – Grundlagen der Softwarezuverlässigkeit und Theoretische Informatik

Algebraic Systems of Fixpoint Equations over Semirings: Theory and Applications

Maximilian Schlund

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr. H. Seidl
Prüfer der Dissertation:	
	1. Prof. Dr. Dr. h.c. F. J. Esparza Estaun
	2. Prof. Dr. M. Holzer, Justus-Liebig-Universität Giessen

Die Dissertation wurde am 20.08.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 14.01.2016 angenommen.

Abstract

Algebraic systems of fixpoint equations $X = F(X)$ arise in various areas of computer science, such as program verification, deductive databases, and formal language theory. In this thesis we study algorithms for solving algebraic systems based on Newton's method as proposed by (Esparza, Kiefer, Luttenberger, 2010). We first investigate the theoretical properties and algorithmics of Newton's method on semirings. Then we present FPSOLVE, an efficient and generic implementation of various methods for solving algebraic systems over semirings. Exploring various applications, we (1) propose general representations for the provenance of recursive Datalog programs, (2) study the efficient computation of the subword closure of context-free languages, and (3) show how statistical parsing of natural languages can be improved using ideas from the combinatorics underlying Newton's method.

Kurzzusammenfassung

Algebraische Systeme von Fixpunktgleichungen der Form $X = F(X)$ finden Anwendung in verschiedenen Bereichen der Informatik, wie Programmverifikation, deduktive Datenbanken und die Theorie der formalen Sprachen. In dieser Arbeit beschäftigen wir uns mit Algorithmen zur Lösung von algebraischen Systemen, welche auf dem Newtonverfahren basieren, wie von (Espraza, Kiefer, Luttenberger, 2010) vorgeschlagen. Hierbei untersuchen wir zuerst die theoretischen und algorithmischen Eigenschaften des Newtonverfahrens auf Semiringen. Anschließend stellen wir FPSOLVE vor, eine effiziente und generische Implementierung verschiedener Methoden zur Lösung algebraischer Systeme. Abschließend erkunden wir diverse Anwendungen. Wir (1) präsentieren eine allgemeine Darstellung der Provenienz von rekursiven Datalog-Programmen, (2) untersuchen die effiziente Berechnung des Teilwortabschlusses von kontextfreien Sprachen und (3) zeigen, wie statistisches Parsen von natürlicher Sprache mit Ideen der kombinatorischen Analyse des Newtonverfahrens verbessert werden kann.

Acknowledgments

First and foremost I want to thank my advisor Prof. Javier Esparza for introducing me to fascinating problems and for always taking the time to discuss progress and non-progress. His inspiring personality and his style of research were a constant source of motivation. I am especially grateful for his clever insights that stimulated my research and for giving me the freedom to explore my own ideas.

I am especially indebted to Michael Luttenberger – many results in this theses originated from joint work with him. I have learned so much about mathematics, music, and of course etymology and Frankish history from him. Thanks for these last four years and for the constant encouragement (Lemma 2).

Very special thanks go to Prof. Markus Holzer for agreeing to review this thesis.

Thanks to my great colleagues (current and former) at chair I7, especially to my favorite office-roommate René, Stefan, Andreas (thanks for the encouragement during the last weeks!), Jan, Christian, Christian, Theo, Philipp, Philipp, and Salomon.

I also want to thank Prof. Sándor Fekete who has been an inspiring mentor to me for many years now. Thanks to all the other people from the working group on algorithmic game theory: Jonas, Jano, Jann, and especially Sebastian Wild (for showing me the beauty in generating functions and for very pleasant collaborations).

Thanks to my fellow PhD students at TUM: Flo (for being such a good roommate) and Thomas and Wolf (for the 3 o'clock coffee-breaks, general nerd-talk, and for being great friends).

Many thanks to Michał Terepeta (from whom I learned a lot about C++) and to my student collaborators Michael Kerscher and Georg Bachmeier who all helped with the implementation of FPsolve.
Thanks a lot to Anna for reading my horrible drafts, enduring my mood swings during the last months, and for being my favorite person in this universe – I love you.

Last but not least, I am grateful to my parents for their unconditional love and support.

Contents

1 Introduction

This thesis is concerned with *algebraic systems*. On a purely syntactic level, an algebraic system is a set of fixpoint equations of the form

$$\begin{aligned} X_1 &= F_1(X_1, \ldots, X_n) \\ &\vdots \\ X_n &= F_n(X_1, \ldots, X_n) \end{aligned}$$

where $X_1, \ldots, X_n$ are variables and $F_1, \ldots, F_n$ are polynomials, i.e. formal terms involving multiplication and addition. For example if we fix some variables X, Y and constants a, b, c, d then the following is an algebraic system

$$\begin{aligned} X &= a \cdot X \cdot Y + b \\ Y &= c \cdot X \cdot Y \cdot Y + d \end{aligned}$$

To give a semantics to algebraic systems we have to specify how to interpret the addition and multiplication operation and we have to fix "values" for the constants. For the expressions to be well-defined, these values should live in an algebraic structure that defines two operations (addition and multiplication). Such a natural structure is a *semiring*, which intuitively is a ring without (additive) inverses. Hence, in a semiring we can add and multiply but cannot (necessarily) subtract nor divide.

Fixpoint equations inductively *define* semiring values X and Y. These values can be obtained by repeatedly substituting the defining right hand side for variables:

$$\begin{aligned}
X &= a \cdot X \cdot Y + b \\
&\overset{X}{=} a \cdot (a \cdot X \cdot Y + b) + b = aaXY + ab + b \\
&\overset{Y}{=} aaX(c \cdot X \cdot Y \cdot Y + d) + ab + b = aaXcXYY + aaXd + ab + b \\
&\vdots
\end{aligned}$$

In the limit this process yields X as an infinite sum. This requires that infinite sums are well-defined in our semiring and satisfy certain laws like infinite distributivity[1]. Semirings enjoying this property are called ω-continuous semirings.

Example 1.1. *Simple examples of semirings are*

- *The natural numbers $\mathbb{N}$ together with the usual addition and multiplication.*
- *The set of formal languages 2^{Σ^*} where the addition operation is set-union and multiplication is given by the concatenation of languages.*

Over the natural numbers multiplication is commutative, hence $(\mathbb{N}, +, \cdot, 0, 1)$ is a commutative *semiring. Multiplication of formal languages on the other hand is not commutative:*

$$\{a\} \cdot \{b\} = \{ab\} \neq \{ba\} = \{b\} \cdot \{a\}$$

The semiring of natural numbers is not ω-continuous since the infinite sum $1 + 1 + 1 + \ldots$ is not an element of $\mathbb{N}$. However, $\mathbb{N}$ can be turned into the ω-continuous semiring $\mathbb{N}_\infty$ by adding the special symbol ∞ and defining $\infty + x := \infty$ and $\infty \cdot x = \infty$.
The formal languages on the other hand constitute an ω-continuous semiring $(2^{\Sigma^*}, \cup, \cdot, \emptyset, \{\varepsilon\})$.

Over ω-continuous semirings, algebraic systems inductively define a value for each variable. We call this value the *solution* of the equations. More precisely, since the definition specifies a least-fixpoint computation its value is the *least* solution.

Example 1.2. *Consider the system from before under the interpretation $a = 0.2, b = 0.8, c = d = 0.5$. We interpret the system over the semiring of (extended) nonnegative reals, $\mathbb{R}^{\geqslant 0}_\infty = \mathbb{R}^{\geqslant 0} \cup \{\infty\}$.*

$$\begin{aligned}
X &= a \cdot X \cdot Y + b \\
Y &= c \cdot X \cdot Y \cdot Y + d
\end{aligned}$$

[1] In other words, such infinite sums behave like absolutely convergent series over the real numbers.

This system has several solutions over $\mathbb{R}_\infty^{\geqslant 0}$*:*

$$\begin{aligned}(X, Y) &= (\infty, \infty)\\ (X, Y) &= (1, 1)\\ (X, Y) &= \left(\frac{24}{25}, \frac{5}{6}\right) = (0.96, 0.83\overline{3})\end{aligned}$$

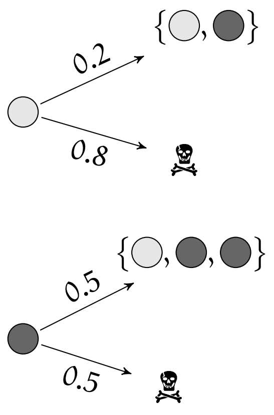
While the first two solutions are trivial, the third one is interesting since it is the least solution of the system (w.r.t. the usual ordering $\leqslant$*) and is the limit of the sequence* $0, F(0), F^2(0), \ldots$*:*

$$(0, 0), \quad \left(\frac{4}{5}, \frac{1}{2}\right), \quad \left(\frac{22}{25}, \frac{3}{5}\right), \quad \ldots$$

1.1 Modeling via Algebraic Systems

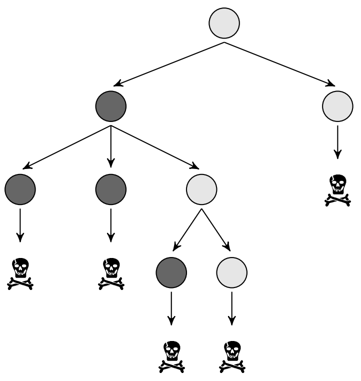
Algebraic systems are commonly not studied out of mathematical curiosity alone but because they help to *model* interesting problems.

Example 1.3. *Consider a population with two types of individuals* ○ *and* ●*. The reproductive behavior of the population is governed by the following probabilistic rules (we can think of the individuals as bacteria performing mitosis and apoptosis).*



For example, an individual ○ *either dies with probability* 0.8 *or procreates by separating into an individual of type* ○ *and one of type* ●*. We further assume that individuals reproduce independently of each other.*

An example generation tree for a population starting out with one individual ○ *is depicted in the following:*



In this history tree, all individuals of the population eventually die. Since individuals reproduce independently, the probability of this "history tree" t is obtained by multiplying the probabilities of the production rules involved which yields

$$\Pr(t) = (0.2)^2 \cdot (0.8)^2 \cdot 0.5 \cdot (0.5)^3 = \frac{1}{625} = 0.0016$$

An interesting question is: what is the total probability that a population eventually dies out if it initially consists of a single individual ○?

This question can be solved by formulating an appropriate algebraic system whose least solution yields the desired probability. To this end we use the law of total probability and the independence assumption to obtain a recursive equation for the probability that the population {○} *eventually dies out:*

$$\Pr(\{○\} \text{ dies out}) = 0.2 \cdot \Pr(\{○\} \text{ dies out}) \cdot \Pr(\{●\}\text{dies out}) + 0.8$$

and similarly for {●}

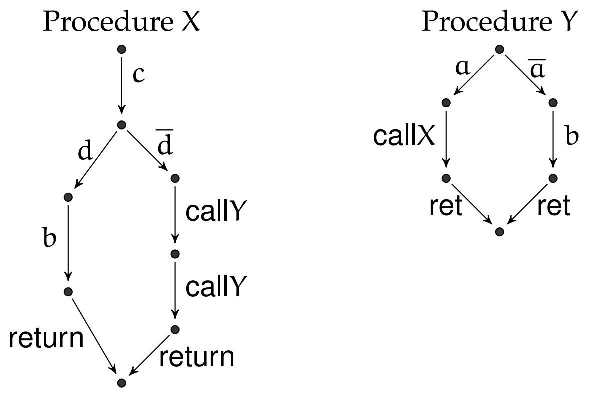
$$\Pr(\{●\} \text{ dies out}) = 0.5 \cdot \Pr(\{○\} \text{ dies out}) \cdot \Pr(\{●\}\text{dies out}) \cdot \Pr(\{●\}\text{dies out}) + 0.5$$

If we define $X := \Pr(\{○\} \text{ dies out})$ *and* $Y := \Pr(\{●\} \text{ dies out})$ *we obtain the familiar system*

$$\begin{aligned} X &= 0.2 \cdot X \cdot Y + 0.8 \\ Y &= 0.5 \cdot X \cdot Y \cdot Y + 0.5 \end{aligned}$$

with least solution $(X, Y) = (0.96, 0.83\overline{3})$. *As we can see, the probability that the population* {○} *eventually dies out is not* 1, *and hence the cumulative probability of all infinite histories is larger than* 0. *The model mentioned here is an example of a Galton-Watson branching process which is well-studied in probability theory (cf. [Har02]).*

Important examples of algebraic systems arise from the static analysis of procedural programs (cf. [EKL10]). In the following illustration we show the call graphs of two mutually recursive procedures X and Y. The edges are labeled with call/return actions and abstract letters representing statements in the program.



If we represent sequential composition of statements by multiplication and branching by addition we can encode the above call graphs as the following algebraic system

$$\begin{aligned} X &= c \cdot (d \cdot b + \overline{d} \cdot Y \cdot Y) \\ Y &= a \cdot X + \overline{a} \cdot b. \end{aligned}$$

Note that up to this point we have not specified a structure over which to interpret the abstract constants $a, b, c, \ldots$. In fact, *different interpretations* of these constants give rise to *different semantics* of the program.

Example 1.4 (Relational interpretation). *Let* V *be the set of program variables appearing in procedures* X *and* Y*. If we fix a domain* D *(e.g. integers), a state of the program is specified by a function* $s : V \to D$*. Thus the set of states* $S := D^V$ *is given by the set of all functions from* V *to* D*. Each statement of the program defines a transformation on the set of states and thus, each program statement corresponds to a relation* $R \subseteq S \times S$*. Sequential execution corresponds to the join of relations and branching corresponds to the union of relations.*

Hence, we interpret statements as elements of the semiring of relations

$$(2^{S \times S}, \cup, \cdot, \emptyset, I)$$

with $X \cdot Y := \{(x, y) : \exists z \in S \; (x, z) \in X \wedge (z, y) \in Y\}$ *and the identity relation* I*. The solution* (X, Y) *of the algebraic system then gives a* summary *(a sound overapproximation) of the effects of procedures* X *and* Y*. This interpretation can be seen as an instance of the more general functional approach to program analysis proposed by Sharir and Pnueli [SP81].*

Example 1.5 (Probabilistic interpretation). *If we interpret the constants* $a, b, c, \ldots$ *as probabilities to execute the respective statement we obtain (using that* $d + \overline{d} = 1 = a + \overline{a}$*)*

$$\begin{aligned} X &= cdb + c(1-d)YY) \\ Y &= aX + (1-a)b. \end{aligned}$$

Similar to the branching process example from before, the solution (X, Y) *of this system gives the probabilities that procedures* X *and* Y *eventually terminate.*

Example 1.6 (Language interpretation)**.** *If we interpret each constant* x *as the singleton language* $\{x\}$ *and further interpret multiplication as concatenation and addition as union we effectively view the algebraic system as a context-free grammar.*

$$\begin{aligned} X &= \{c\} \cdot (\{d\} \cdot \{b\} + \{\overline{d}\} \cdot Y \cdot Y) \\ Y &= \{a\} \cdot X + \{\overline{a}\} \cdot \{b\} \end{aligned}$$

$$\begin{aligned} X &\to cdb \mid c\overline{d}YY \\ Y &\to aX \mid \overline{a}b \end{aligned}$$

Note that the connection between algebraic systems and context-free grammars is well known from the work of Chomsky and Schützenberger [CS63]. We recall this connection more formally in Chapter 3.

The above examples motivate why we study algebraic systems abstractly over general semirings. More importantly, they also motivate the study of generic algorithms (that do not depend on a particular semiring) for solving (resp. approximating) algebraic systems. Since algebraic systems arise in very different applications, such generic algorithms are widely applicable.

1.2 Fixpoint iteration

The classical generic algorithm for solving algebraic systems $x = F(x)$ is fixpoint iteration: given an initial guess x_0 we repeatedly apply the function F to compute the sequence of approximations $x_k := F^k(x_0)$,

$$x_0, F(x_0), F(F(x_0)), F^3(x_0), \ldots$$

until either $x_{k+1} = x_k$ (i.e. we reached a fixpoint) or at least x_k is "sufficiently close" to $F(x_k)$. In mathematics, the Banach fixed-point theorem guarantees that the sequence converges to a fixpoint (for any x_0) if F defines a contraction over a complete metric space X. In computer science, convergence of this sequence over an ω-complete partial order is established by Kleene's fixpoint theorem if F is continuous (and if x_0 is chosen as the least element of the set). In particular, if F is a polynomial over an ω-continuous semiring we obtain from this result that fixpoint iteration converges (in the limit).

However, fixpoint iteration can take a long time to reach a sufficiently close approximation, as shown by Etessami and Yannakakis [EY05] for the nonnegative reals. They consider the following equation with unique solution $X = 1$

$$X = \frac{1}{2}X^2 + \frac{1}{2}.$$

On this example, fixpoint iteration started with $x_0 = 0$ needs at least 2^{k-3} steps to approximate the solution with k bits of precision, i.e. $1 - x_n > 2^{-k}$ for $n < 2^{k-3}$ [EY05].

$$\begin{tikzcd}
\text{Polynomial system over } \mathbb{R}^{\geqslant 0} & \text{Context-free grammar} \\
\begin{array}{l} X = aXX + b \\ \quad a = 0.6, b = 0.4 \end{array} \arrow[r] \arrow[d, "\text{solution}"] & X \to aXX \mid b \arrow[d, "\text{derivation trees}"] \\
X = \sum_{t \in \mathcal{T}_X} \mathbf{val}(t) & \mathcal{T}_X \arrow[l] \\
\text{Solution in } \mathbb{R}^{\geqslant 0} & \text{Set of trees}
\end{tikzcd}$$

Figure 1.1: Illustration of the connection between fixpoint systems and context-free grammars.

As a faster alternative to fixpoint iteration, Etessami and Yannakakis propose to solve such polynomial equations over the nonnegative reals using the well-known Newton's method from calculus starting with $x_0 = 0$.

1.3 Algebraic Systems, Context-Free Grammars, and Newton's Method

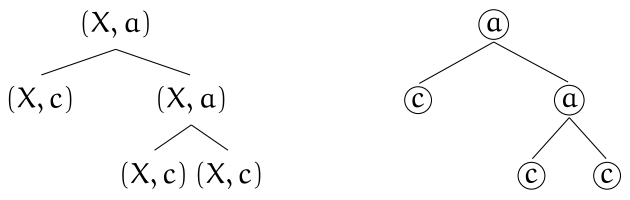
Roughly speaking, a single fixpoint equation $x = \sum_{i=0}^{n} m_i(x)$ corresponds to a grammar rule of the form $x \to m_0 \mid \cdots \mid m_n$ and a system of equations corresponds to a set of such rules, viz. a context-free grammar. The formal relationship between grammars and equations is depicted in Figure 1.1.

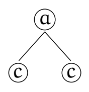
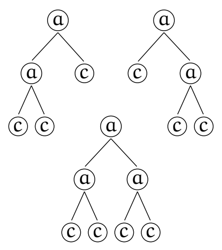
Given a derivation tree of this associated grammar, we define its *valuation* as the product of its leaves in left-to-right order. It is well-known [Boz99, EKL10] that the solution of an fixpoint system is given as the sum of (the valuations of) *all* derivation trees of the associated grammar. Hence, we can approximate the solution to a fixpoint equation using any method that systematically explores (and evaluates) this set of derivation trees.

Fixpoint iteration can be seen as a particular method to enumerate derivation trees. By induction we can show that fixpoint iteration corresponds to evaluating the derivation trees of the associated grammar by increasing height (cf. [EKL07a, EKL10]).

Example 1.7. *Consider the equation* $X = aXX + c$ *with an interpretation over the real numbers with* $a = 0.6$ *and* $b = 0.4$*. The associated context-free grammar is* $X \to aXX \mid c$*. In the* *following table we depict the derivation trees of* G *up to height* 2 *and for each height we also record the sum of the yields of the trees up to that height (once un-interpreted and once evaluated in* $\mathbb{R}$*).*

Since in this example every production is labeled by a unique alphabet symbol we can draw a derivation trees (on the left) in an isomorphic fashion as shown on the right.

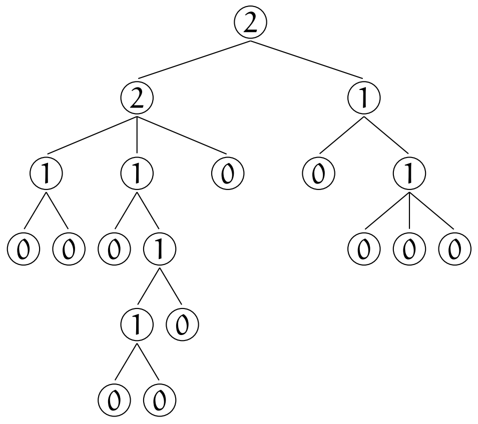


Height	*Trees*	$\sum_{t:\text{height}(t)\leqslant H} Y(t)$	$h\left(\sum_{t:\text{height}(t)\leqslant H} Y(t)\right)$
0	ⓒ	c	0.4
1		$c + acc$	$\frac{62}{125} = 0.496$
2		$c + acc + aaccc$ $+acacc + aaccacc$	≈ 0.565

In [EKL07a, EKL10], Esparza, Kiefer, and Luttenberger showed that Newton's method can be interpreted as a procedure for enumerating derivation trees w.r.t. *tree dimension*.

The dimension of a tree can be inductively defined in a bottom-up fashion as the maximum dimension of its children (*plus one*, if this maximum is attained at least twice). For the base case, we define the dimension of a leaf as 0.

Example 1.8. *The following tree is of height* 5 *and has dimension* 2*. The number in each node indicates the dimension of the subtree rooted at that node.*



The concept of tree dimension was originally discovered by Strahler [Str52] and is thus often called Strahler number in combinatorics. We refer to Section 3.6 for a brief survey (see also [ELS14a]).

Note that the set of trees of a particular dimension is in general infinite (as opposed to the set of trees of a particular height). This infinite set however has a very simple structure which allows us to evaluate it effectively. More specifically, we can *unfold* a given context-free grammar G into a linear grammar $G^{<d}$ that generates exactly the derivation trees of G having dimension less than d (cf. Section 3.4.1). For each dimension d, the unfolded grammar $G^{<d}$ can be interpreted as a *linear* system of equations which are easier to solve than the original nonlinear system over many semirings. In particular, over commutative semirings this system is right-linear of the form $x = Ax + b$ and can be explicitly solved by means of the Kleene star as $x = A^*b$.

1.4 Applications of Algebraic Systems

Datalog is a deductive language (a strict subset of Prolog) which is used to specify and query databases. A Datalog program is a set of Horn clauses which can also be interpreted as an algebraic system. In databases, *provenance* describes additional information attached to facts which can be used to trace the origin of query results. Green et al. [GKT07] showed how provenance analyses can be reduced to solving algebraic systems over a suitable semiring.

Formal and Natural Languages The connection between algebraic systems and context-free grammars is well-known from the work of Chomsky and Schützenberger [CS63]. We review this connection in Section 3.2 and use it in Chapter 7 to study the subword closure of context-free languages.

A "weighted" grammar formalism widely used in natural language processing (NLP) are so called probabilistic context-free grammars (PCFGs) which are essentially algebraic systems over $[0, 1]$. An important property of PCFGs for natural language is *consistency*, i.e. the cumulative probability of all finite derivations should be equal to 1. Etessami and Yannakakis have shown how to check consistency of PCFGs in polynomial time [EY09]. Surprisingly, several examples of grammars used in NLP fail to be consistent [WE07].

1.5 Contributions and Outline

In this thesis we follow three major lines of work concerning algebraic systems and Newton's method:

- We contribute to the theoretical foundations of Newton's method. We define Newton's method via grammar unfolding, analyze its convergence behavior, and combinatorics of tree dimension.
- We devise efficient algorithms and data structures for Newton's method and develop FPsolve, a generic implementation of Newton's method over semirings.
- We explore applications in Datalog provenance, formal language theory, and natural language processing.

Overview of Technical Contributions

1. Our theoretical contributions are described in Chapter 3. Most of our results were published in [LS13, LS15].

   In Section 3.5 we study the convergence behavior of Newton's method and prove a general theorem on the convergence speed of Newton's method over commutative semirings.

   In Section 3.4 we suggest an improved definition of Newton's method via dimension-unfolding. We relate this unfolding to the usual definition via derivatives and show how to obtain a closed formulation of Newton's method over noncommutative semirings, in particular matrix semirings with commutative entries.

   In Section 3.6 we contribute to the combinatorial theory underlying Newton's method. We prove a tight relation between pathwidth and tree dimension, settling an open question by David Eppstein.

2. In Chapter 4 we study the algorithmic aspects of Newton's method and describe our implementation FPsolve. Chapter 5 is devoted to the study of algorithms and data structures for semilinear sets. Semilinear sets are important as the (algebraic) elements of the "most general" commutative idempotent semiring. Most of these results were published in [STL13, ELS14b, ELS15]:

3. We explore three different applications which use theoretical and algorithmic ideas from the study of Newton's method. Most of these results were published in the following papers [BLS15, LS13, LS15, LS14, SLE14].

a) In Chapter 6 we propose concisely stored regular expression as a general, finite representation for provenance of recursive Datalog programs. We describe how to compute these expressions via Newton's method and show how to simplify them over special semirings. We generalize the work of Green et al. [GKT07] and improve the work of Deutch et al. [DMRT14].

b) In Chapter 7 we study the descriptional complexity of finite automata and regular expressions for representing the subword closure of context-free grammars. We settle two open questions of Gruber, Holzer, and Kutrib [GHK09], prove new complexity results, and describe an application to grammar testing.

c) In Chapter 8 we suggest tree dimension as a measure for syntactic complexity and briefly show how statistical parsing of natural language can be improved by enriching grammars with dimension information.

2 Preliminaries

Here we fix some basic notions and definitions used throughout the thesis.

2.1 Basics: Numbers, Languages, Orders, Fixpoints

We denote the empty set by $\emptyset$. The set of all functions $f : Y \to X$ is denoted by X^Y. For a set X, we denote its powerset by 2^X (i.e. we identify a subset with its characteristic function $f : X \to \{0, 1\}$). For a relation $R \subseteq X \times X$ over a set X, we denote its reflexive, transitive closure by R^*. We denote the natural numbers (non-negative integers) by $\mathbb{N} = \{0, 1, 2, \dots\}$ and furthermore we define the natural numbers extended by a symbol for infinity as $\mathbb{N}_\infty := \mathbb{N} \cup \{\infty\}$. For any k, the first $k + 1$ natural numbers form the set $\mathbb{N}_k := \{n \in \mathbb{N} : n \leqslant k\}$. Finally, for $n \in \mathbb{N}$ we set $[n] := \{k \in \mathbb{N} : 1 \leqslant k \leqslant n\}$, in particular note that $[0] = \emptyset$. For a finite set X we denote the set of all its k-element subsets by $\binom{X}{k} := \{S \subseteq X : |S| = k\}$. If $|X| = n$ we have that $\left|\binom{X}{k}\right| = \binom{n}{k} = \frac{n!}{(n-k)!k!}$. We write $\mathbb{R}$ for the real numbers, $\mathbb{R}^{\geqslant 0}$ for the nonnegative reals, and $\mathbb{R}^{\geqslant 0}_\infty := \mathbb{R}^{\geqslant 0} \cup \{\infty\}$ for the nonnegative real numbers extended by a symbol for infinity.

An *alphabet* Σ is a finite set. A (finite) word w of length n over Σ is a sequence of n elements from Σ (i.e. formally a function from $[n]$ to Σ). We write $w = a_1 \dots a_n$ with $a_i \in \Sigma$ for such a word. We denote the length of a word w by $|w|$ and for $a \in \Sigma$ we write $|w|_a = |\{i \in [|w|] : w_i = a\}|$ for the number of a's appearing in w. The *empty word* is the (unique) word of length 0 and is denoted by ε. Given two words $u = a_1 \dots a_n$ and $v = b_1 \dots b_m$ we define their *concatenation* as $u \cdot v := a_1 \dots a_n b_1 \dots b_m$. This definition extends canonically to sets of words: For two languages A, B over a common alphabet Σ we define their concatenation as the language $A \cdot B := \{u \cdot v : u \in A, v \in B\}$.

For a language A, we inductively define $A^0 := \{\varepsilon\}$ and $A^{n+1} := A \cdot A^n$. This allows us to define the *Kleene star* of A as $A^* := \bigcup_{i \geqslant 0} A^i$. Furthermore, we denote by $A^{\leqslant k}$ all words from A of length at most k, i.e. $A^{\leqslant k} := \bigcup_{0 \leqslant i \leqslant k} A^i$.

A relation $\leqslant$ on a set X is called a *partial order* if it is reflexive, transitive, and antisymmetric, i.e.

- $x \leqslant x$,
- $x \leqslant y \wedge y \leqslant z \Longrightarrow x \leqslant z$,
- $x \leqslant y \wedge y \leqslant x \Longrightarrow x = y$.

The resulting tuple $(X, \leqslant)$ is called a partially ordered set (as usual, we just write X if the partial order is clear). Given a subset $A \subseteq X$ we call an element $x \in X$ the *least upper bound* (or *supremum*) of A if $\forall a \in A : a \leqslant x$ and for every other $y \in X$ also satisfying $\forall a \in A : a \leqslant y$ we have $x \leqslant y$. Not every subset has a supremum, but if it does, then by the antisymmetry of $\leqslant$ the supremum is unique and we denote it by $\sup_{\leqslant} A$ (omitting the reference to the partial order $\leqslant$ if it is clear)

We call a countable, non-decreasing sequence in X an *ω-chain*. Put formally, $(x_i)_{i \in \mathbb{N}} \in X$ is an ω-chain, if $x_i \leqslant x_{i+1}$ $\forall i \in \mathbb{N}$.

Let $(X, \leqslant)$ be a partial order with a least element $\bot \in X$. We call $(X, \leqslant, \bot)$ an *ω-complete partial order* (ω-**cpo**), if every ω-chain $(x)_{i \in \mathbb{N}}$ has a supremum.

A function $f : (X, \leqslant, \bot) \to (Y, \sqsubseteq, \bot')$ mapping an ω-**cpo** $(X, \leqslant, \bot)$ to an ω-**cpo** $(Y, \sqsubseteq, \bot')$ is called *monotone*, if $x \leqslant y \Longrightarrow f(x) \sqsubseteq f(y)$. $f : (X, \leqslant, \bot) \to (Y, \sqsubseteq, \bot')$ is called ω-continuous, if for every ω-chain $(x)_{i \in \mathbb{N}}$ with $\sup_{\leqslant}\{x_i : i \in \mathbb{N}\} = x$ we have $\sup_{\sqsubseteq}\{f(x_i) : i \in \mathbb{N}\} = f(x)$ (in particular, this supremum exists).

Theorem 2.1 (Kleene's fixpoint theorem [Kui97]). *Every ω-continuous function $f : (X, \leqslant, \bot) \to (Y, \sqsubseteq, \bot')$ over a ω-**cpo** has a least fixpoint x and*

$$x := \sup\{f^i(\bot) : i \in \mathbb{N}\}.$$

2.2 Semirings, Formal Power Series, Polynomials, Matrices, Vectors

A *monoid* $(M, \cdot, 1)$ is a set M together with an associative binary operation "$\cdot$" and a neutral element 1, i.e. the following conditions are satisfied:

- $\cdot : M \times M \to M$,
- $\forall x, y, z \in M : (x \cdot y) \cdot z = x \cdot (y \cdot z)$,

- $\forall x \in M : x \cdot 1 = 1 \cdot x = x$.

We simply write M to refer to the monoid, if the operation and neutral element are clear from the context. A monoid M is *commutative*, if $x \cdot y = y \cdot x$ holds for all $x, y \in M$. Simple examples of (commutative) monoids are the natural numbers with the usual addition operation $(\mathbb{N}_\infty, +, 0)$ or the extended natural numbers with the minimum operation $(\mathbb{N}_\infty, \min, \infty)$.

For a set Σ the *free monoid* generated by Σ is the set Σ^* with $\cdot$ being the concatenation of words. The *free commutative monoid* over Σ is denoted by $\Sigma^\oplus$ and is Σ^* modulo the set of congruences $\forall x, y \in \Sigma : x \cdot y = y \cdot x$. The set of functions $\mathbb{N}^\Sigma$ together with the pointwise addition on $\mathbb{N}$ is a commutative monoid, the neutral element (which we write as 0) being the function that maps every $a \in \Sigma$ to 0.

A monoid homomorphism $h : M \to M'$ is a mapping from a monoid $(M, \cdot, 1)$ to a monoid $(M', \otimes, 1')$ that satisfies

- $h(1) = 1'$
- $h(x \cdot y) = h(x) \otimes h(y)$

An important example of a monoid homomorphism is the *Parikh image* P of a word $w \in \Sigma^*$, $\mathsf{P} : (\Sigma^*, \cdot, \varepsilon) \to (\mathbb{N}^\Sigma, +, 0)$ defined as the (unique) homomorphism satisfying $\mathsf{P}(a) = \delta_a(x)$ with

$$\delta_a(x) = \begin{cases} 1, & \text{if } x = a \\ 0, & \text{otherwise} \end{cases} \qquad \text{for all } a \in \Sigma$$

It is easy to see (e.g. by induction on $|w|$) that the Parikh image just counts the number of all letters appearing in w, i.e. $\mathsf{P}(w)(a) = |w|_a$ for $w \in \Sigma^*, a \in \Sigma$.

A *semiring* $(S, +, \cdot, 0, 1)$ is a set S together with an addition $(+)$ and multiplication $(\cdot)$ operation such that

- $(S, +, 0)$ is a commutative monoid and $(S, \cdot, 1)$ is a monoid.
- Multiplication distributes over addition, i.e. $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$ and $(y + z) \cdot x = (y \cdot x) + (z \cdot x)$ for all $x, y, z \in S$.
- 0 is an annihilator for the multiplication, i.e $0 \cdot x = x \cdot 0 = 0$ for all $x \in S$.

We call a semiring S *commutative* if the monoid $(S, \cdot, 1)$ is commutative. S is called *idempotent* if $x + x = x$ holds for all $x \in S$. A semiring is *finite*, if S is a finite set. As usual, we write s^n for the n-fold product $s^n := \prod_{i=1}^n s$ and additionally we write $n \cdot s$ to denote the n-fold sum $\sum_{i=1}^n s$.

Examples of commutative semirings are again the natural numbers with the usual addition and multiplication $(\mathbb{N}, +, \cdot, 0, 1)$ or the idempotent *tropical semiring* over the (extended) natural $(\mathbb{N}_\infty, \min, +, \infty, 0)$ or real numbers $(\mathbb{R}_\infty^{\geqslant 0}, \min, +, \infty, 0)$, respectively. A standard example of a non-commutative (but idempotent) semiring is the semiring of formal languages with union and concatenation of languages $(2^{\Sigma^*}, \cup, \cdot, \emptyset, \{\varepsilon\})$.

A semiring S is called *naturally ordered*, if the relation $\sqsubseteq$ defined by $x \sqsubseteq y \iff \exists z : x + z = y$ is a partial order. We call a semiring ω-continuous if it is naturally ordered and countably infinite sums are defined and behave like absolutely convergent series, formally:

Definition 2.2 (ω-continuous semiring [Ési08, EKL07a]). *A naturally ordered semiring* $(S, +, \cdot, 0, 1)$ *is* ω-continuous *if the following conditions hold*

1. *For all sequences* $(a_i)_{i\in\mathbb{N}} \in S$, $\sum_{i\in\mathbb{N}} a_i := \sup\{\sum_{0\leqslant i\leqslant n} a_i : n \in \mathbb{N}\}$ *exists.*
2. *The order of summation is irrelevant for infinite sums, i.e. for all partitions* $\{I_j : j \in J\}$ *of* $\mathbb{N}$ *we have* $\sum_{i\in\mathbb{N}} a_i = \sum_{j\in J} \sum_{i\in I_j} a_i$.
3. *The distributive laws also hold for infinite sums, i.e. for all* $c \in S$ *we have* $\sum_{i\in\mathbb{N}}(c \cdot a_i) = c \cdot \left(\sum_{i\in\mathbb{N}} a_i\right)$ *and* $\sum_{i\in\mathbb{N}} a_i \cdot c = \left(\sum_{i\in\mathbb{N}} a_i\right) \cdot c$.

In ω-continuous semirings we can define the *Kleene star* of an element $a \in S$ by

$$a^* := \sum_{i\in\mathbb{N}} a^i$$

The following lemma collects identities that hold in any ω-continuous semiring [1].

Lemma 2.3 ([DKV09, ÉK04]). *Let S be an* ω*-continuous semiring and* $x, y \in S$ *then*

1. $a^* = 1 + aa^*$
2. $(ab)^* = 1 + a(ba)^*b$
3. $(a + b)^* = (a^*b)^*a^*$

Given two semirings $(R, +_R, \cdot_R, 0_R, 1_R), (S, +_S, \cdot_S, 0_S, 1_S)$ a semiring *homomorphism* is a map $h : R \to S$ such that for all $x, y \in R$

- $h(0_R) = 0_S$ and $h(1_R) = 1_S$,
- $h(x +_R y) = h(x) +_S h(y)$,
- $h(x \cdot_R y) = h(x) \cdot_S h(y)$.

As remarked in [Lut10], a homomorphism h between two ω-continuous semirings is ω-continuous if and only if $h\left(\sum_{i\in\mathbb{N}} a_i\right) = \sum_{i\in\mathbb{N}} h(a_i)$ for all $a_i \in S$.

A mapping r from a monoid M to a semiring S is called a *formal power series* over M with coefficients in S. Such mappings are commonly written as $r = \sum_{m\in M}(r, m)m$

[1] More generally, these identities hold in any inductive *-semiring, e.g. the semiring of regular or context-free languages which are both not ω-continuous.

where $(r, m) \in S$ is the image of $m \in M$, called the *coefficient* of m. The set of all such formal power series is denoted by $S\langle\langle M\rangle\rangle$. For a power series $r \in S\langle\langle M\rangle\rangle$ we define its *support* as elements in M with non-zero coefficients: $\mathrm{supp}(r) = \{m \in M : (r, m) \neq 0\}$. If the support of r is finite, we call r a *polynomial*. The set of all polynomials over M with coefficients in S is denoted by $S\langle M\rangle$.

To be able to define a semiring structure on $S\langle\langle M\rangle\rangle$ we have to make either some additional assumptions on the semiring, e.g. that arbitrary (i.e. also non-countable) sums are defined, or restrict the monoid M (cf. [DKV09]). Here, we follow the approach of Ésik [Ési08] and require that M is *finitely decomposable*, i.e. every element $m \in M$ can be written as a product $m = x \cdot y$ only in a finite number of ways. The set of formal power series over a locally finite monoid M with coefficients in a semiring S can be endowed with a semiring structure by defining for $r = \sum_{m\in M}(r, m)m$ and $s = \sum_{m\in M}(s, m)m$ the addition component-wise

$$r + s := \sum_{m\in M} ((r, m) + (s, m))m$$

and multiplication via the *Cauchy product*

$$r \cdot s := \sum_{m\in M} \left(\sum_{\substack{x,y\in M \\ xy=m}} (r, x) \cdot (s, y) \right) m.$$

It is well-known that $S\langle\langle M\rangle\rangle$ is an ω-continuous semiring if S is [DKV09].

A power series is called a *polynomial* if it has finite support. We write $S\langle M\rangle$ for the set of polynomials over M with coefficients in S. Note that $S\langle M\rangle$ is also a semiring, but in general not ω-continuous since countable sums of polynomials need not be a polynomial anymore. In particular, the Kleene star cannot be defined in $S\langle M\rangle$. However, in special cases, e.g. when S is finite, $S\langle M\rangle$ is ω-continuous.

Of particular interest to us is the semiring $\mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle$ of formal power series (resp. polynomials) over the free monoid Σ^* with coefficients in the extended natural numbers $\mathbb{N}_\infty$. This semiring is "free" in the sense that every valuation $h : \Sigma \to S$ of alphabet symbols into an ω-continuous semiring S extends uniquely to a homomorphism $\hat{h} : \mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle \to S$. Similarly, the semiring $\mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle$ of power series in commuting variables with coefficients in $\mathbb{N}_\infty$ can be seen as *the* commutative semiring freely generated by Σ.

Given two finite index sets I, J the set of $I \times J$ matrices over a semiring is $S^{I\times J}$, i.e. a matrix with entries in a semiring S is a mapping from $I\times J$ to S. We write $A = (a_{i,j})_{i\in I, j\in J}$ and $A_{i,j} = a_{i,j}$. For a matrix $A \in S^{I\times J}$ its *transpose* $A^T \in S^{J\times I}$ is defined by $A^T_{i,j} := A_{j,i}$ for $i \in I, j \in J$.

The set of square matrices $S^{I\times I}$ is a semiring with the usual matrix addition and multiplication:

$$(A + B)_{i,j} := A_{i,j} + B_{i,j} \quad \text{for } i, j \in I$$

$$(A \cdot B)_{i,j} := \sum_{k \in I} A_{i,k} \cdot B_{k,j} \quad \text{for } i, j \in I$$

As for power series, $S^{I \times I}$ is an ω-continuous semiring if S is.

Vectors over S indexed by an index set I are simply functions from S^I. Given a matrix $A \in S^{I \times J}$ and a vector $b \in S^J$, the product $A \cdot b$ is the vector

$$(A \cdot b)_i := \sum_{j \in J} a_{i,j} \cdot b_j \quad \text{for } i \in I$$

Similarly, for $A \in S^{I \times J}$ and a vector $b \in S^I$ the product $b \cdot A$ is defined by

$$(b \cdot A)_j := \sum_{i \in I} b_i \cdot a_{i,j} \quad \text{for } j \in J$$

2.3 Automata and Formal Language Theory

2.3.1 Regular Languages

A *(nondeterministic) finite automaton (NFA)* $\mathcal{A}$ is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. We write $q \xrightarrow{a} q'$ if $q' \in \delta(q, a)$. The transition function extends naturally to a function $\hat{\delta} : Q \times \Sigma^*$ via the inductive definition for $q \in Q, a \in \Sigma, w \in \Sigma^*$:

$$\begin{aligned} \hat{\delta}(q, \varepsilon) &:= \{q\}, \\ \hat{\delta}(q, aw) &:= \bigcup_{q' \in \delta(q,a)} \hat{\delta}(q', w). \end{aligned}$$

Abusing notation, we use δ as well to denote this extension of the transition function to $Q \times \Sigma^*$, and write $q \xrightarrow{w} q'$ if $q' \in \delta(q, w)$ as well.

An ε-NFA is an NFA where we also allow the empty word as a transition label. It is well-known that every ε-NFA can be transformed into an (ε-free) NFA over the same set of states (albeit with a quadratic increase in the number of transitions). Thus, we also allow transitions of an NFA to carry the label ε.

The *language* accepted by an NFA $\mathcal{A}$ is defined as $\mathcal{L}(\mathcal{A}) := \{w \in \Sigma^* : \delta(q_0, w) \cap F \neq \emptyset\}$. We call an NFA *deterministic* (or DFA) if $|\delta(q, a)| = 1$ for all $q \in Q, a \in \Sigma$. Hence, for DFAs we can view δ as a function $\delta : Q \times \Sigma^* \to Q$.

The set of *regular* (or *rational*) *expressions* REG_Σ over an alphabet Σ is the set of terms inductively defined by:

- $\varepsilon \in \mathsf{REG}_\Sigma$, $\mathbf{a} \in \mathsf{REG}_\Sigma$ for every $a \in \Sigma$.
- If $r, s \in \mathsf{REG}_\Sigma$ then $(r + s) \in \mathsf{REG}_\Sigma$ and $(r \cdot s) \in \mathsf{REG}_\Sigma$.

- If $r \in \mathsf{REG}_\Sigma$ then $(r)^* \in \mathsf{REG}_\Sigma$.

As usual, we omit parentheses and assume that multiplication binds stronger than addition to improve readability (furthermore we often write multiplication implicitly by juxtaposition).

The *language* of a regular expression is inductively defined by

- $\mathcal{L}(\varepsilon) := \{\varepsilon\}$ and $\mathcal{L}(a) := \{a\}$ for every $a \in \Sigma$.
- $\mathcal{L}(r + s) := \mathcal{L}(r) \cup \mathcal{L}(s)$ and $\mathcal{L}(r \cdot s) := \mathcal{L}(r) \cdot \mathcal{L}(s)$.
- $\mathcal{L}(r^*) := \mathcal{L}(r)^*$.

Kleene's theorem [DKV09] states that for a finite alphabet Σ,

- the set of languages over Σ defined by regular expressions and
- the set of languages over Σ accepted by NFAs

coincide. Hence, we call this set the *regular languages*.

2.3.2 Context-Free Languages

A context-free grammar (CFG) is a tuple $G = (\mathcal{X}, \Sigma, \to)$ with

- $\mathcal{X}$ is a finite set of *nonterminals* (or *variables*),
- Σ is a finite alphabet of *terminals*,
- $\to \subseteq \mathcal{X} \times (\mathcal{X} \cup \Sigma)^*$ is a set of *(production) rules*, and

When writing down the set of rules of a grammar, we often collect all productions of the form $X \to r_i$ for a nonterminal X and write $X \to r_1 \mid r_2 \mid \ldots \mid r_k$. A CFG is *linear*, if every rule is of the form $X \to vYw$ or $X \to w$ for some $v, w \in \Sigma^*$. We call a CFG *ε-free*, if $(A, \varepsilon) \notin \to$ for all $A \in \mathcal{X}$.

The *size* of a CFG G, denoted by $|G|$, is defined as the total sum of lengths of the right-hand-sides of all rules. An easy observation is that for $L := \max\{\sum_{r: x \to r} |r| : x \in \mathcal{X}\}$ the maximal length of all right-hand-sides and $n := |\mathcal{X}|$, we have $|G| \leqslant n \cdot L$.

Note that in our definition of CFGs we have not fixed a particular start symbol. The elements of $(\mathcal{X} \cup \Sigma)^*$ are called *sentential forms* and the rules of the grammar induce a *derivation relation* $\Rightarrow_G$ over sentential forms. This relation $\Rightarrow_G \subseteq (\mathcal{X} \cup \Sigma)^* \times (\mathcal{X} \cup \Sigma)^*$ is defined by: If $X \to \beta$ for some $\beta \in (\mathcal{X} \cup \Sigma)^*$ then $\alpha X \gamma \Rightarrow_G \alpha\beta\gamma$ for all $\alpha, \gamma \in (\mathcal{X} \cup \Sigma)^*$.

We call a sequence $\alpha_i \in (\mathcal{X} \cup \Sigma)^*$ with $X \Rightarrow_G \alpha_1 \Rightarrow_G \cdots \Rightarrow_G \alpha_n$ a *derivation* of α_n from $X \in \mathcal{X}$. If $\alpha_n \in \Sigma^*$ we call it a *terminal derivation*. More concisely, we can express this derivation via the reflexive transitive closure $\Rightarrow_G^*$ of the derivation relation: $X \Rightarrow_G^* \alpha_n$. We define the *language* of a nonterminal $X \in \mathcal{X}$ as $\mathcal{L}(X) := \{w \in \Sigma^* : X \Rightarrow_G^* w\}$. At times we explicitly fix a start symbol S which allows us to talk about the language of a CFG $\mathcal{L}(G) := \mathcal{L}(S)$.

We slightly divert from the usual definition of *derivation trees* of a CFG G by requiring that each node is labeled with a production rule of the grammar.

Definition 2.4 (Labels, derivation tree, yield). *For a node labeled with (X, α), we call X the* label *of the node and write $\mathsf{lab}(t)$ for the set of all labels of a tree t. The label of a tree is the label of its root node. Let G be a CFG,*

- *A single node v labeled with (X, α) is a derivation tree of G if $X \to \alpha$ is a rule in G. Its yield is defined as $Y(v) := \alpha$.*
- *The tree t with subtrees $t_1, \ldots, t_k$ is a derivation tree if*
    - *$t_1, \ldots, t_k$ are derivation trees with labels $X_1, \ldots, X_k$,*
    - *t is labeled with (X, α) where $\alpha = w_0 X_1 w_1 X_2 \cdots w_{k-1} X_k w_k$ (with $w_i \in \Sigma^*$),*
    - *$X \to \alpha$ is a rule of G.*

  Its yield is $Y(t) := w_0 Y(t_1) w_1 \cdots w_{k-1} Y(t_k) w_k$.

We call t a *partial* derivation tree if $Y(t) \notin \Sigma^*$ (i.e. there is at least one nonterminal in its yield). A partial tree is a *pump tree* if $Y(t)$ contains exactly one nonterminal X which is also the label of the root of t. If $Y(t) \in \Sigma^*$, we call t a *terminal* derivation tree.

We denote the set of all terminal derivation trees of a grammar G that are labeled with X by $\mathcal{T}_X^G$. In the following, we reserve the name "derivation trees" for terminal trees, and explicitly state when we mean partial trees.

A context-free grammar is in *2-normal form* (2NF) if the right hand side of every production has length *at most* two, i.e. $\to \subseteq \mathcal{X} \times (\Sigma \cup \mathcal{X})^{\leqslant 2}$. It is well-known that every CFG G can be transformed into a CFG G' in 2NF (by introducing fresh variables to binarize all rules) such that (1) there is a bijection between the sets of derivation trees of both grammars, and (2) that there is a constant c (independent of G) such that $|G'| \leqslant c \cdot |G|$ [LL09].

While 2NF is very simple, one often has to consider rules of the form $A \to aB$ or $A \to Ba$ as special cases. Hence we often use a more restricted normal form that forbids such rules: A CFG is in *canonical two form* (C2F) if $\to \subseteq \mathcal{X} \times \left(\{\varepsilon\} \cup \Sigma \cup \mathcal{X} \cup \mathcal{X}^2\right)$. This normal form has appealing properties (cf. [Ben77]):

1. Every grammar G can be transformed into a grammar G' in C2F with at most linear blowup in size.
2. The transformation is ambiguity preserving, i.e. there is a bijection between the set of derivation trees of G and G'.
3. If G is ε-free then G' is ε-free.

The transformation into C2F is again carried out by introducing fresh variables to binarize all rules and eliminate terminals from quadratic rules. Note that despite being at most linear, the increase in size can be practically relevant. Unfortunately, finding the smallest binarization of a CFG is an NP-hard problem [GR14].

3 Theory of Algebraic Systems

In this chapter we present our theoretical results concerning Newton's method for solving algebraic systems over semirings.

We first review the connections between algebraic systems and context-free grammars in Section 3.2. In particular, we recall that the solution of an algebraic system can be viewed as the ambiguity function of the associated grammar. In Section 3.3 we review how fixpoint iteration can be viewed combinatorially as an enumeration method for derivation trees by height.

Next, in Section 3.4 we present two equivalent definitions of Newton's method: First, a combinatorial definition based on a grammar unfolding by tree dimension (aka. Strahler number) and a definition of Newton's method via derivatives that can be related to the usual definition of Newton's method over $\mathbb{R}$.

In Section 3.4.4 we study Newton's method in the non-commutative setting. Specifically, we show how to obtain a closed form for the Newton approximations as regular tree expressions over non-commutative semirings, and how to compute Newton's method over the (non-commutative) matrix semirings with commutative entries.

Our main technical contribution is the convergence result for Newton's method over commutative semirings in Section 3.5.2 and the consequence that Newton's method converges in finite time over all commutative k-collapsed semirings.

Finally, in Section 3.6 we study the tree dimension in greater detail and prove a new connection to the pathwidth of a tree in 3.6.2 which answers an open question by David Eppstein. Furthermore, we show how to apply the combinatorial properties of tree dimension for the optimization of arithmetic expressions in Section 3.6.3.

Most of these results were published in [LS13, LS15] apart from the study of Newton's method in the non-commutative setting (especially for matrices) and the combinatorial optimization of arithmetic expressions.

3.1 Polynomial Systems

In the following we fix an ω-continuous semiring S and a finite set of variables $\mathcal{X} = \{X_1, \dots, X_n\}$. Informally speaking, a polynomial system over S is a set of defining equations

$$\begin{aligned} X_1 &= F_1(X_1, \dots, X_n) \\ &\vdots \\ X_n &= F_n(X_1, \dots, X_n) \end{aligned}$$

where $F_1, \dots, F_n$ are "polynomial expressions" built from elements of S and variables in $\mathcal{X}$ and a "solution" of such system is any valuation (replacing the variables with elements in S) such that the equations are satisfied.

More precisely, we define a *monomial* m as a (non-empty) finite product term $m = a_0 \cdot X_{i_1} \cdot a_1 X_{i_1} \cdots a_{k-1} \cdot X_{i_k} \cdot a_k$ with $a_i \in S$. A *polynomial expression* p over S is a finite sum of monomials $p = \sum_{i=1}^{N} m_i$ with $N \geqslant 1$. We write $\mathsf{Pol}_S^{\mathcal{X}}$ for the set of polynomial expressions over S with variables in $\mathcal{X}$. Note that formally the "product" and "sum" are uninterpreted function symbols and hence polynomial expressions do not have an interpretation yet. The interpretation is given by viewing a polynomial expression p as a function: For $p \in \mathsf{Pol}_S^{\mathcal{X}}$ s.t. $p = \sum_{j=1}^{N} m_j$ we define the associated polynomial function $\hat{p} : S^{[n]} \to S$ by $\hat{p}(s) := \sum_{j=1}^{N} h(m_j)$, where h is the evaluation homomorphism given by $h(X_i) = s_i$, $h(a) = a$ for $a \in S$ and $h(x \cdot y) = h(x) \cdot h(y)$ for subterms x, y of a monomial. A *polynomial system* over S in variables $\mathcal{X}$ is a finite set of equations $X_i = p_i$ where $X_i \in \mathcal{X}, p_i \in \mathsf{Pol}_S^{\mathcal{X}}$ for all $i \in [n]$. A *solution* of such a system is a vector $s \in S^{[n]}$ such that $s_i = \hat{p_i}(s)$ for all $i \in [n]$.

It is well-known that the associated polynomial function of a polynomial expression over S is ω-continuous (cf. [Kui97]) and hence, every polynomial system has a least solution in $S^{[n]}$.

Note that we strictly distinguish "polynomial expressions" $\mathsf{Pol}_S^{\mathcal{X}}$ which are terms, from "polynomials" with coefficients in S, i.e. elements from $S\langle \mathcal{X}^* \rangle$ which are functions with finite support from $\mathcal{X}^*$ to S. Intuitively, the difference is that in a polynomial expression the semiring coefficients can appear both in front of and in between variables:

Example 3.1. *Consider the semiring S of formal languages over the alphabet $\Sigma = \{a, b, c\}$, i.e. $S = (2^{\Sigma^*}, \cup, \cdot, \emptyset, \{\varepsilon\})$. Let $\mathcal{X} = \{X\}$, then the polynomial expression $p = \{a\} \cdot X \cdot \{b\} + \{c\}$ defines the function $\hat{p}(X) = \{a\} \cdot X \cdot \{b\} + \{c\}$ with least fixpoint $L = \{a^{n+1} c b^{n+1} : n \in \mathbb{N}\}$.*

However, the only polynomial $q \in S\langle \mathcal{X}^* \rangle$, *i.e.* $q = \sum_{i \geqslant 0} c_i \cdot X^i$ *such that* q *has the least fixpoint* L *is the trivial* $q = L \cdot X^0$. *This can be seen easily since* $L^i \cap (\Sigma^* \setminus L) \neq \emptyset$ *for all* $i > 1$, *so we must have* $c_i = 0$ *for all* $i > 0$.

3.2 From Equations to Grammars and Back

Here we survey the well-known relations between systems of equations and context-free grammars. We recall the fact that the least solution of an algebraic system is equal to the ambiguity function of the associated grammar, which counts the number of derivation trees for a given word. Hence, to approximate solutions of algebraic system it suffices to approximate the set of derivation trees of a context-free grammar.

Polynomial Equations and Algebraic Systems The above example justifies why we introduce "polynomial expressions" instead of just viewing polynomial systems as a vector of polynomials from $S\langle \mathcal{X}^* \rangle$.

However, to obtain a unified presentation of the theory we can move to a richer structure and view a polynomial system over S as a vector of polynomials from $\mathbb{N}_1\langle (\mathcal{X} \cup \Sigma)^* \rangle$ together with an interpretation function $\iota : \Sigma \to S$, where Σ are new symbolic names replacing the semiring constants in the polynomial system. Hence, from a theoretical point of view it is sufficient to consider so called "algebraic systems" over $\mathbb{N}_\infty\langle\langle \Sigma^* \rangle\rangle$ if we want to study polynomial systems over some S. Also note that since our polynomial expressions consist only of non-empty monomials, the corresponding polynomials are even members of $\mathbb{N}_1\langle (\mathcal{X} \cup \Sigma)^+ \rangle$, i.e. the coefficient of the empty monomial is 0.

Similarly to [Sal90] we define an *algebraic system* in variables $\mathcal{X} = \{X_1, \ldots, X_n\}$ over $\mathbb{N}_\infty\langle\langle \Sigma^* \rangle\rangle$ as a vector of polynomials $p_1, \ldots, p_n \in \mathbb{N}_1\langle (\Sigma \cup \mathcal{X})^* \rangle$. It is important to note that we only allow coefficients in $\mathbb{N}_1 = \{0, 1\}$ for the polynomials.

A *solution* of such an algebraic system is a vector r of power series $r = (r_1, \ldots, r_n)$, $r_i \in \mathbb{N}_\infty\langle\langle \Sigma^* \rangle\rangle$ satisfying the system, i.e. for the *evaluation homomorphism* $h : \mathbb{N}_\infty\langle (\Sigma \cup \mathcal{X})^* \rangle \to \mathbb{N}_\infty\langle\langle \Sigma^* \rangle\rangle$ induced by $h(X_i) = r_i$ we have $r_i = h(p_i) \forall i \in [n]$.

It is well-known [Boz99] (see also [MW67]) that we can associate with any polynomial system $X_1 = p_1, \ldots, X_n = p_n$ over S an algebraic system over $\mathbb{N}_\infty\langle\langle \Sigma^* \rangle\rangle$ such that the least solution of the polynomial system can be obtained by applying the homomorphism $h : \mathbb{N}_\infty\langle\langle \Sigma^* \rangle\rangle \to S$ (induced by the interpretation $\iota : \Sigma \to S$) to the least solution of the algebraic system. Figure 3.1 illustrates this result.

We furthermore define an "algebraic system over an ω-continuous semiring S" as an algebraic system $p_1, \ldots, p_n \in \mathbb{N}_1\langle (\Sigma \cup \mathcal{X})^* \rangle$ together with an interpretation function $\iota : \Sigma \to S$. A *solution* of this system over S is a solution $r_1, \ldots, r_n$ of the algebraic system (over $\mathbb{N}_\infty\langle\langle \Sigma^* \rangle\rangle$) together with a vector of elements $s_1, \ldots, s_n \in S$ such that $h(r_i) = s_i$ where h is the unique homomorphism $h : \mathbb{N}_\infty\langle\langle \Sigma^* \rangle\rangle \to S$ induced by ι. For a fixed

$$\begin{tikzcd}
\text{Polynomial system over } S & \text{Algebraic system over } \mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle \\
\begin{array}{rcl} X & = & XsY+t \\ Y & = & YYt+sX+t \end{array} \arrow[r, "\text{un-interpret}", "\iota(a)=s,\iota(b)=t"'] \arrow[d, "\text{least solution}"', "(s_X,s_Y)\text{ in } S"] & \begin{array}{rcl} X & = & XaY+b \\ Y & = & YYb+aX+b \end{array} \arrow[d, "\text{least solution}"] \\
\begin{array}{rcl} s_X & = & \sum_{w\in\{a,b\}^*} c_{X,w}\cdot h(w) \\ s_Y & = & \sum_{w\in\{a,b\}^*} c_{Y,w}\cdot h(w) \end{array} & \begin{array}{rcl} X & = & \sum_{w\in\{a,b\}^*} c_{X,w}\cdot w \\ Y & = & \sum_{w\in\{a,b\}^*} c_{Y,w}\cdot w \end{array} \arrow[l, "\text{re-interpret}"', "\iota\rightsquigarrow h"] \\
\text{Solution: \textit{element} of } S & \text{Solution: \textit{power series} in } \mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle
\end{tikzcd}$$

Figure 3.1: Illustration of the connection between polynomial and algebraic systems.

interpretation, the polynomials $p_1, \dots, p_n$ define functions on S via the evaluation homomorphisms $h : \mathcal{X}^* \to S$ which we denote by $\hat{p}_i$ as before.

Algebraic Systems and Grammars It is well-known from the work of Chomsky and Schützenberger, that algebraic systems are closely related to context-free grammars [CS63]. Every algebraic system $p_1, \dots, p_n \in \mathbb{N}_1\langle(\Sigma \cup \mathcal{X})^*\rangle$ induces a context-free grammar $G := (\mathcal{X}, \Sigma, \to)$ where $\to$ is defined by: If $(\alpha, p_i) \neq 0$ then $X_i \to \alpha$. Since the coefficients of the p_i are in $\mathbb{N}_1 = \{0, 1\}$ this correspondence even yields a bijection between algebraic systems and CFGs (as remarked already in [Sal90]).

The *ambiguity* of a CFG G w.r.t. a nonterminal X is a mapping giving the number of derivation trees labeled with X that yield w for each word $w \in \Sigma^*$, i.e.

$$\mathsf{amb}_{G,X}(w) := \left|\{t \in \mathcal{T}_X^G : \mathsf{Y}(t) = w\}\right|$$

is a formal power series in $\mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle$. Given an algebraic system over $\mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle$, it is well-known that its least solution is equal to the *ambiguity* of the corresponding grammar G. Since $\mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle$ is the ω-continuous semiring freely generated by Σ, the solution to any algebraic system over some ω-continuous semiring S is obtained via mapping amb to S via the unique homomorphism $h : \mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle \to S$ induced by the interpretation $\iota : \Sigma \to S$ [EKL10, Boz99]. Figure 3.2 depicts this connection schematically.

Similar to amb, the *commutative ambiguity* of a CFG G counts for every word w the number of derivation trees that yield a permutation of w, i.e.

$$\mathsf{camb}_{G,X}(w) := \left|\{t \in \mathcal{T}_X^G : \mathsf{P}(\mathsf{Y}(t)) = \mathsf{P}(w)\}\right|,$$

$$\begin{tikzcd}
\text{Algebraic system over } \mathbb{N}_\infty\langle\!\langle\Sigma^*\rangle\!\rangle & & \text{Context-free grammar} \\
\begin{array}{rcl} X & = & XaY + b \\ Y & = & YYb + aX + b \end{array} \arrow[rr] \arrow[d, "\text{least solution}"] & & \begin{array}{rcl} X & \rightarrow & XaY \mid b \\ Y & \rightarrow & YYb \mid aX \mid b \end{array} \arrow[d, "\text{count trees}"] \\
\begin{array}{rcl} X & = & \sum_{w\in\{a,b\}^*} c_{X,w} \cdot h(w) \\ Y & = & \sum_{w\in\{a,b\}^*} c_{Y,w} \cdot h(w) \end{array} & \dfrac{X = \mathsf{amb}_X}{Y = \mathsf{amb}_Y} & \begin{array}{rcl} \mathsf{amb}_X & = & \sum_{w\in\{a,b\}^*} c_{X,w} \cdot h(w) \\ \mathsf{amb}_Y & = & \sum_{w\in\{a,b\}^*} c_{Y,w} \cdot h(w) \end{array} \\
\text{Solution in } \mathbb{N}_\infty\langle\!\langle\Sigma^*\rangle\!\rangle & & \text{Ambiguity in } \mathbb{N}_\infty\langle\!\langle\Sigma^*\rangle\!\rangle
\end{tikzcd}$$

Figure 3.2: Illustration of the connection between algebraic systems and context-free grammars.

where P denotes the Parikh image (cf. Section 2.2). Let us illustrate these results by means of a longer example.

Example 3.2. *Consider the simple equation over the semiring $\mathbb{R}_\infty^{\geqslant 0}$ from Example 1.7 in the introduction*

$$X = 0.6XX + 0.4$$

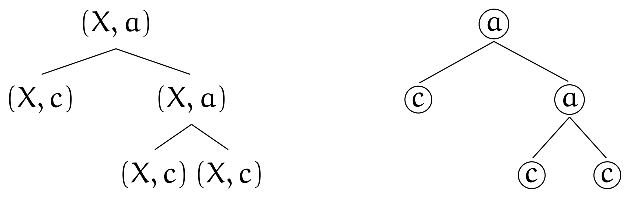
which corresponds to the algebraic "system"

$$X = aXX + c$$

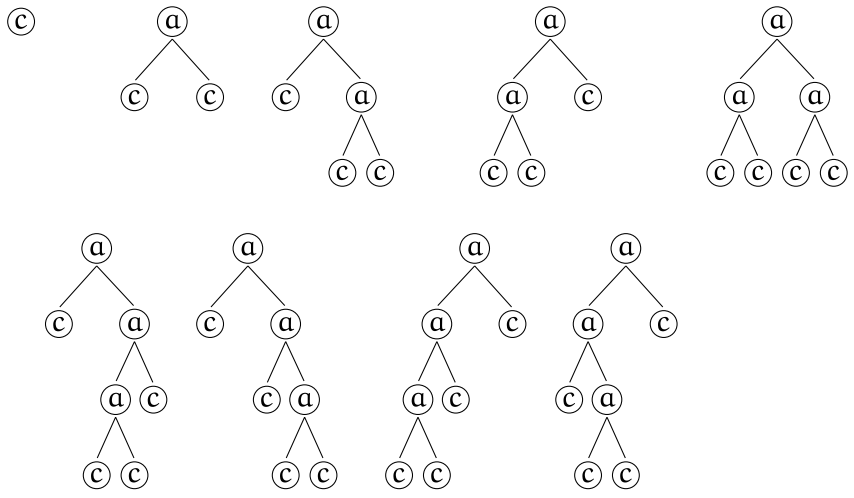
with $\iota(a) = 0.6, \iota(c) = 0.4$. The algebraic system in turn corresponds to a context-free grammar over $\Sigma = \{a, c\}$ in the obvious way:

$$X \rightarrow aXX \mid c.$$

Since this grammar is unambiguous, we have $\mathsf{amb}_{G,X}(w) \in \{0, 1\}$. For example $\mathsf{amb}_{G,X}(acc) = 1$ and $\mathsf{amb}_{G,X}(cc) = 0$. For unambiguous grammars like these, each word in $\mathcal{L}(X)$ encodes exactly one derivation tree. In our case this is a binary tree with inner nodes labeled with a and leaves labeled with c. For instance the word $acacc$ describes the derivation tree shown below on the left. Recall from the Example 1.7 that we can draw the derivation tree on the left in an isomorphic fashion as shown on the right since every rule in the grammar is labeled with a unique terminal.



To determine camb*, we have to count for each* $w \in \Sigma^{\oplus}$ *how many derivation trees yield* w *modulo commutativity. For* $n \in \mathbb{N}$*, every tree with* n *inner nodes has* $n + 1$ *leaves, so* $\mathsf{camb}(w) \neq 0$ *if and only if* $w = a^n c^{n+1}$ *for some* $n \in \mathbb{N}$*. Up to* $n \leqslant 3$ *inner nodes, it is still easy to enumerate all possible derivation trees:*



As we can see, the numbers of trees with n *inner nodes are (for* $n = 0, 1, 2, 3$*)*

$$1, 1, 2, 5, \ldots$$

Hence, the first terms of the power series camb *are*

$$\mathsf{camb}_{G,X} = 1 \cdot c + 1 \cdot acc + 2 \cdot a^2c^3 + 5 \cdot a^3c^4 + \ldots.$$

In general, the number of binary trees with n *inner nodes is the* n*-th Catalan number [GKP89]* C_n *given by*

$$C_n := \frac{1}{n+1}\binom{2n}{n} \quad \text{for } n \in \mathbb{N}.$$

Therefore

$$\mathsf{camb}_{G,X}(a^n c^{n+1}) = \frac{1}{n+1}\binom{2n}{n}.$$

We can use the fact that camb *is the least solution of the algebraic system* $X = aXX + c$ *over* $\mathbb{N}_\infty\langle\langle\Sigma^{\oplus}\rangle\rangle$ *to obtain the solution of*

$$X = 0.6 \cdot X \cdot X + 0.4$$

over the real numbers. To this end we apply the (unique) homomorphism $h : \mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle \to \mathbb{R}$ *induced by the interpretation* $\iota(a) = \frac{3}{5}, \iota(c) = \frac{2}{5}$ *to our solution* camb*:*

$$
\begin{aligned}
X = h(\mathsf{camb}_{G,X}) &= \sum_{n=0}^{\infty} \frac{1}{n+1}\binom{2n}{n}\left(\frac{3}{5}\right)^n\left(\frac{2}{5}\right)^{n+1} \\
&= \frac{2}{5}\sum_{n=0}^{\infty} C_n\left(\frac{6}{25}\right)^n = \\
&= \frac{2}{5}\sum_{n=0}^{\infty} C_n\left(\frac{6}{25}\right)^n = \\
&= \frac{2}{5}C\left(\frac{6}{25}\right)
\end{aligned}
$$

with $C(z)$ *the generating function of the Catalan numbers, which is well-known [GKP89]:*

$$C(z) = \frac{1-\sqrt{1-4z}}{2z}.$$

So we can continue to get

$$X = \frac{2}{5}\cdot\frac{1-\sqrt{1-\frac{24}{25}}}{\frac{12}{25}} = \frac{2}{5}\cdot\frac{20}{12} = \frac{2}{3}$$

as the least solution to the equation $X = 0.6 \cdot X \cdot X + 0.4$ *over* $\mathbb{R}$.

Note that the purpose of the previous example is not to compute the solution of a quadratic equation in the most complicated way imaginable but rather to illustrate that solving (resp. approximating) the solution of algebraic systems over semirings reduces to computing (resp. approximating) the ambiguity function amb of the associated grammar. Similarly, it suffices to study camb if the semiring is commutative.

3.3 Fixpoint Iteration

Fixpoint iteration is the straightforward translation of Kleene's fixpoint theorem into an approximation method. To approximate the least fixpoint of an algebraic system $p = (p_1, \dots, p_n)$ over an ω-continuous semiring S, fixpoint iteration computes the sequence $0, \hat{p}(0), \hat{p}^2(0), \dots, \hat{p}^n(0) \in S$ until either a fixpoint is reached (i.e. $\hat{p}^n(0) = \hat{p}^{n-1}(0)$) or the approximation is deemed sufficient.

By induction on the number of fixpoint iterations we can easily show that fixpoint iteration on an algebraic system corresponds to evaluating the yields of the derivation trees of the associated context-free grammar by increasing height (cf. [EKL07a, EKL10]).

Moreover, we can *unfold* the grammar G into a grammar $G^{<h}$ whose derivation trees are in one-to-one correspondence with those derivation trees of G having height less than h. The following definition is a slightly improved version of the one given in [EL11]. It is inspired by the semi-naive evaluation strategy for Datalog programs (cf. [AHV95]).

Definition 3.3 (Height unfolding). *Given a context-free grammar* $G = (\mathcal{X}, \Sigma, \rightarrow)$ *the* height unfolding *of* G *is the sequence of grammars* $G^{<H} = (\mathcal{X}_H, \Sigma, \rightarrow_H), H \in \mathbb{N}$ *defined by:*

- *If* $X \in \mathcal{X}$ *then* $X^{=h} \in \mathcal{X}_H$ *for all* $0 \leqslant h \leqslant H-1$ *and* $X^{<h} \in \mathcal{X}_H$ *for all* $1 \leqslant h \leqslant H$.
- *If* $X \rightarrow \alpha \in \Sigma^*$ *then* $X^{=0} \rightarrow_H \alpha$.
- $X^{<h} \rightarrow_H X^{=h-1}$ *for all* $1 \leqslant h \leqslant H$ *and* $X^{<h} \rightarrow_H X^{<h-1}$ *for all* $2 \leqslant h \leqslant H$.
- *If* $X \rightarrow w_0 X_1 w_1 \cdots w_k X_k w_{k+1}$ *with* $w_i \in \Sigma^*$ *then we have for all* $1 \leqslant j \leqslant k$ *and* $0 \leqslant h \leqslant H-1$

$$X^{=h} \rightarrow_H w_0 X_1^{<h} w_1 \cdots X_{j-1}^{<h} w_{j-1} X_j^{=h-1} w_j X_{j+1}^{<h-1} \cdots w_k X_k^{<h-1}$$

The intuition is that $X^{<h}$ can only produce derivation trees of height less than h and similarly, $X^{=h}$ produces only derivation trees of height exactly h. The unfolding is then obtained by distinguishing the cases how to obtain a tree of height h: Let t be a derivation tree of height (exactly) h labeled with X and having subtrees $t_1, \ldots, t_k$ labeled with $X_1, \ldots, X_k$, respectively. Then there exists exactly one rule in the above unfolding that generates the given subtrees since the rule is uniquely determined by the (label of the) rightmost subtree having height $h-1$.

More formally, we can obtain the following result, e.g. by structural induction on t (see also [EL11]).

Theorem 3.4. *Let* G *be a CFG and* $G^{<H}$ *for some* $H \in \mathbb{N}$ *be its height unfolding. Every derivation tree* t *of* G *of height less than* H *with root label* $X \in \mathcal{X}$ *corresponds to exactly one derivation tree* $\hat{t}$ *of* $G^{<H}$ *with root label* $X^{<H} \in \mathcal{X}_H$. *Moreover, the yields of* t *and* $\hat{t}$ *coincide,* $Y(t) = Y(\hat{t})$.

3.4 Newton's Method

In the same way as fixpoint iteration can be viewed as a systematic procedure for enumerating derivation trees by height, we can define Newton's method as a procedure for enumerating derivation trees by a different tree parameter, called the *dimension* (or *Strahler number*). We review this definition in Section 3.4.1 and relate it to the more familiar definition via derivatives in Section 3.4.2. Then in Section 3.4.3 we recall how Newton's method can be simplified over commutative semirings. In particular we provide a link to Newton's method over the real numbers. Finally in Section 3.4.4 we show

how to give a closed formulation of Newton's method over non-commutative semirings and in particular for the semirings of matrices with commuting entries.

Most results reviewed here are well-known from [EKL07a, EKL07b, EL11, LS13, LS15] and especially M. Luttenbergers thesis [Lut10]. One contribution is to make the connection to Newton's method over $\mathbb{R}$ explicit which has appeared implicitly in the previous work. The other contribution is the study of Newton's method for non-commutative semirings and in particular for the matrix semirings with commuting entries.

Since our goal is to define Newton's method as an enumeration of trees by *dimension* we first define the notion of tree dimension and give an easy combinatorial characterization. For a rooted tree we define its dimension inductively

Definition 3.5 (Dimension). *Let t be a rooted tree with subtrees $t_1, \dots, t_k$. The* dimension *of t is defined inductively as follows:*

- *If t is a single node (i.e. $k = 0$) we set $\dim(t) := 0$.*
- *Otherwise, let $D := \max\{\dim(t_i) : i \in [k]\}$ and set*

$$\dim(t) := \begin{cases} D + 1, & \text{if } \exists i \neq j : D = \dim(t_i) = \dim(t_j) \\ D, & \text{otherwise} \end{cases}$$

Hence, the dimension of a tree is simply the maximum dimension of its children (*plus one*, if this maximum is attained at least twice).

Remark 3.6. *The name "dimension" stems from the fact that trees of dimension $d > 1$ can be naturally drawn in $\mathbb{R}^d$ (cf. [Lut10]).*

The following useful characterization of the tree dimension is used in Section 3.5.2 to prove our main convergence result for Newton's method. We call a binary tree of height h *perfect* if it has 2^h leaves. A rooted tree t' is a *minor* of a rooted tree t if t' can be obtained from t by edge contractions[1].

Lemma 3.7. *Let t be a rooted tree. Then $\dim(t)$ is the maximum height of a perfect binary tree that is a minor of t.*

Proof. We show the statement by induction on the structure of t. If t is a leaf the statement trivially holds. Otherwise, let $\dim(t) = D$; we distinguish two cases:

1) t has (exactly) one subtree t_i of dimension D and all other subtrees have dimension at most $D - 1$:

   By induction, the highest perfect binary embeddable into t_i has height D and is also a minor of t. Moreover, the highest perfect binary tree minors of all other subtrees have height at most $D - 1$, so the highest binary tree minor of t is the one found in t_i.

[1]Contracting an edge (u, v) means to identify vertices u and v.

2) t has at least two subtrees t_i, t_j of dimension $D - 1$ and all other subtrees have dimension at most $D - 1$:

   Again, by induction we can embed perfect binary trees of height $D - 1$ into at least two subtrees and hence t possesses a perfect binary tree minor of height D. Moreover, we cannot find a higher binary tree minor since the highest minors of all subtrees inductively have height at most $D - 1$.

$\square$

Corollary 3.8. *For any tree* T *we have* $\dim(T) \leqslant h(T)$.

3.4.1 Definition of Newton's Method as Dimension-Unfolding

Similar to the height-unfolding, we can unfold every CFG G into a grammar $G^{<d}$ such that there is a bijection between the derivation trees of $G^{<d}$ and the derivation trees of G having dimension less than d. For simplicity, we define the unfolding first for grammars in C2F, i.e. each rule has the form $A \to a$, $A \to \varepsilon$, $A \to BC$, or $A \to B$. Note that this is not really a restriction since every CFG can be transformed into C2F preserving the ambiguity of the grammar (and with at most linear blowup).

Definition 3.9 (Dimension unfolding – C2F, [LS15]). *Given a context-free grammar* $G = (\mathcal{X}, \Sigma, \to)$ *in C2F the* dimension unfolding *of* G *is the sequence of grammars* $G^{<D} = (\mathcal{X}_D, \Sigma, \to_D), D \in \mathbb{N}$ *defined by:*

- *If* $X \in \mathcal{X}$ *then* $X^{=d} \in \mathcal{X}_D$ *for all* $0 \leqslant d \leqslant D - 1$ *and* $X^{<d} \in \mathcal{X}_D$ *for all* $1 \leqslant d \leqslant D$.
- *If* $X \to x$ *with* $x \in \Sigma \cup \{\varepsilon\}$ *then* $X^{=0} \to_D x$.
- $X^{<d} \to_D X^{=d-1}$ *for all* $1 \leqslant d \leqslant D$ *and* $X^{<d} \to_D X^{<d-1}$ *for all* $2 \leqslant d \leqslant D$.
- *If* $X \to Y$ *with* $Y \in \mathcal{X}$ *then for all* $0 \leqslant d \leqslant D - 1$ *we have* $X^{=d} \to_D Y^{=d}$.
- *If* $X \to YZ$ *with* $Y, Z \in \mathcal{X}$ *then for all* $1 \leqslant d \leqslant D - 1$ *we have*

$$X^{=d} \to_D Y^{=d}Z^{<d} \mid Y^{<d}Z^{=d} \mid Y^{=d-1}Z^{=d-1}.$$

We can observe that every unfolded grammar $G^{<D}$ is of size $\mathcal{O}(D \cdot |G|)$.

Example 3.10. *Unfolding the grammar* $X \to XX + c$ *w.r.t. dimension produces the following grammar* $G^{<D}$ *for* $1 \leqslant d < D$.

$$\begin{array}{lcl}
X^{=0} & \to & c \\
X^{<1} & \to & X^{=0} \\
X^{<d+1} & \to & X^{<d} \mid X^{=d} \\
X^{=d} & \to & X^{<d}X^{=d} \mid X^{=d}X^{<d} \mid X^{=d-1}X^{=d-1}
\end{array}$$

Similarly to the height-unfolding, we have that the dimension unfolding partitions the set of all derivation trees of G w.r.t. their dimension:

Theorem 3.11 ([LS15]). *Let* G *be a CFG in C2F. For every derivation tree* t *of* G *with* $\dim(t) < d$ *having root label* X *there is exactly one derivation tree* $\hat{t}$ *of* $G^{<d}$ *having root label* $X^{<d}$. *Moreover, the yields of* t *and* $\hat{t}$ *are the same,* $Y(t) = Y(\hat{t})$.

A (fairly technical) proof by induction can be found in [LS15]. To unfold general grammars of arbitrary arity we can either transform them into C2F and use the above unfolding or define the unfolding directly. This definition is a slightly improved version of the one given in [EL11] which produces unfoldings of size $\Omega(2^k)$ for grammars of arity k.

Definition 3.12 (Dimension unfolding – general case). *Given a context-free grammar* $G = (\mathcal{X}, \Sigma, \rightarrow)$ *the* dimension unfolding *of* G *is the sequence of grammars* $G^{<D} = (\mathcal{X}_D, \Sigma, \rightarrow_D)$, $D \in \mathbb{N}$ *defined by:*

- *If* $X \in \mathcal{X}$ *then* $X^{=d} \in \mathcal{X}_D$ *for all* $0 \leqslant d \leqslant D-1$ *and* $X^{<d} \in \mathcal{X}_D$ *for all* $1 \leqslant d \leqslant D$.
- *If* $X \rightarrow \alpha \in \Sigma^*$ *then* $X^{=0} \rightarrow_D \alpha$.
- $X^{<d} \rightarrow_D X^{=d-1}$ *for all* $1 \leqslant d \leqslant D$ *and* $X^{<d} \rightarrow_D X^{<d-1}$ *for all* $2 \leqslant d \leqslant D$.
- *If* $X \rightarrow \alpha_0 X_1 \alpha_1 \cdots X_n \alpha_n$ *with* $\alpha_i \in \Sigma^*$ *is a rule in* G, *we add for all* $1 \leqslant j < k \leqslant n$ *the following rules:*

$$X^{=d} \rightarrow_D \prod_{i=1}^{j-1} \alpha_{i-1} X_i^{<d} \cdot \alpha_{j-1} X_j^{=d} \alpha_j \cdot \prod_{i=j+1}^{n} X_i^{<d} \alpha_i \qquad \text{and}$$
$$X^{=d} \rightarrow_D \prod_{i=1}^{j-1} \alpha_{i-1} X_i^{<d} \cdot \alpha_{j-1} X_j^{=d-1} \alpha_j \prod_{i=j+1}^{k-1} X_i^{<d-1} \alpha_{k-1} \cdot X_k^{=d-1} \alpha_k \prod_{i=k+1}^{n} X_i^{<d-1} \alpha_i.$$

As we remarked before, this general definition is not strictly necessary since we can transform every grammar G of arity k into C2F with linear increase in size and then apply the unfolding to this grammar in C2F resulting in an unfolded grammar of size $\left|G^{<D}\right| \in \mathcal{O}(k \cdot D \cdot |G|)$. On the other hand, if we apply the general unfolding scheme to a grammar of arity k directly, the unfolding $G^{<D}$ has size $\left|G^{<D}\right| \in \Theta(k^2 \cdot D \cdot |G|)$. This mismatch seems strange at first, but note that even the general unfolding scheme is not as compact as it could be – in fact one can also reach a linear increase in size as well by compacting the grammar further using sharing. However, most of the time it is both easier and more efficient to first transform grammars into C2F.

Definition 3.13 (Newton Approximations). *Given an algebraic system* $F = (f_1, \ldots, f_n)$ *with* $f_i \in \mathbb{N}_1\langle(\Sigma \cup \mathcal{X})^*\rangle$, *with an interpretation* $\iota : \Sigma \rightarrow S$, *we define its* d*-th* Newton approximation ν_d *as the function* $\nu_d : \mathcal{X} \rightarrow S$ *given by*

$$\nu_d(X) := h(\mathsf{amb}_{G^{<d}, X}).$$

Here, h is the unique homomorphism $h : \mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle \to S$ induced by ι as defined in the previous Section 3.2. For the sake of readability, we do not explicitly indicate the dependency of ν_d on the system F. If the semiring S is commutative, we can work with the commutative ambiguity camb instead, i.e. $\nu_d(X) = h'(\mathsf{camb}_{G^{<d},X})$ with $h' : \mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle \to S$.

Unfolding Algebraic Systems Applying the natural translation from the context-free grammars to algebraic systems over $\mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle$ we can also define the dimension-unfolding of an algebraic system over a semiring S directly. Let $F = (f_1, \dots, f_n)$ with $f_i \in \mathbb{N}_1\langle(\Sigma \cup \mathcal{X})^*\rangle$ and an appropriate interpretation $\iota : \Sigma \to S$. Informally, we can obtain the unfolded algebraic system from the unfolded grammar in a purely syntactic way: replace "$\to$" by "$=$" and "$|$" by "$+$". We denote the so obtained algebraic system by $F^{<d}$. Let $r_i^{(d)} \in \mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle$ be the $X_i^{<d}$-component of the solution of the unfolded system $F^{<d}$. The d-th Newton approximation ν_d is then given by

$$\nu_d(X_i) = h(r_i^{(d)}),$$

(with $h : \mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle \to$ the homomorphism induced by the interpretation ι).

3.4.2 Newton's Method via Derivatives

From the definition via unfolding we can obtain a form of the d-th Newton approximation of a polynomial system F that looks more like the familiar definition of Newton's method over the real numbers. The benefit of this form is that it is better suited for an algorithmic treatment, as it works directly on the polynomial expressions and also avoids the explicit construction of the unfolding. For this we first need the definition of derivatives of polynomial expressions.

Definition 3.14 (Derivative, Jacobian). *Let $m = a_0X_1a_1 \dots a_kX_ka_{k+1}$ be a monomial with $X_i \in \mathcal{X}$. The* derivative *of m w.r.t. $X \in \mathcal{X}$ is defined as the following polynomial expression in variables $\mathcal{X} \cup \{\hat{X}\}$:*

$$\frac{\partial m}{\partial X} := \sum_{\substack{i \in [k] \\ X_i = X}} a_0X_1a_1 \cdots a_{i-1}\hat{X}a_i \cdots a_kX_ka_{k+1}$$

Accordingly, the derivative of a polynomial expression $f = \sum_{i=1}^{l} m_i$ is defined as

$$\frac{\partial f}{\partial X} := \sum_{i=1}^{l} \frac{\partial m_i}{\partial X}.$$

For a vector of polynomial expressions $F = (f_1, \dots, f_n)$ over variables $X_1, \dots, X_n$, the Jacobian *of F is the $n \times n$ matrix collecting all derivatives, i.e.*

$$J_{i,j} := \frac{\partial f_i}{\partial X_j}$$

Combinatorially, the derivative of a polynomial w.r.t. some variable X is the following procedure: For each monomial containing X, we choose one X that gets "marked" as $\hat{X}$ and we sum over all the choices. Formally we can define this "marking" as a variable substitution: For a monomial $m = a_0 X_1 a_1 \dots a_k X_k a_{k+1}$ we define $m[X_i \leftarrow \hat{X}]$ as the monomial where the variable X_i has been substituted by $\hat{X}$ (note that we explicitly number the variables, so X_i refers to a single variable).

Example 3.15. *Consider the polynomials $f_1 = aXXX + Xb$ and $f_2 = aXbY + YbY$ Then we have*

$$\frac{\partial f_1}{\partial X} = a\hat{X}XX + aX\hat{X}X + aXX\hat{X} + \hat{X}b \qquad \text{and} \qquad \frac{\partial f_1}{\partial Y} = 0$$

Note how the marking remembers the position of the chosen X. If we assume commutative multiplication, then we obtain the more familiar definition: the derivative is the linear approximation of f_1 (in variable $\hat{X}$) with slope $3aX^2$:

$$\frac{\partial f_1}{\partial X} = 3 \cdot aX^2 \cdot \hat{X}.$$

Finally, the Jacobian of the system (f_1, f_2) is

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial X} & \frac{\partial f_1}{\partial Y} \\ \frac{\partial f_2}{\partial X} & \frac{\partial f_2}{\partial Y} \end{pmatrix} = \begin{pmatrix} a\hat{X}XX + aX\hat{X}X + aXX\hat{X} + \hat{X}b & 0 \\ a\hat{X}bY & aXb\hat{Y} + \hat{Y}bY + Yb\hat{Y} \end{pmatrix}$$

Note that $\frac{\partial}{\partial X}$ can be viewed a function mapping polynomial expressions from $\mathsf{Pol}_S^{\mathcal{X}}$ to $\mathsf{Pol}_S^{\mathcal{X} \cup \{\hat{X}\}}$. Also note that the order in which we apply the derivative w.r.t. variables in $\mathcal{X}$ does not matter. Now we are ready to relate the grammar unfolding to the familiar form of Newton's method. For simplicity, we assume (w.l.o.g.) our system F to be in C2F, s.t. each expression f_i is of the form

$$f_i = \sum_{XY \in Q(X_i)} XY + \sum_{X \in L(X_i)} X + \gamma^{(i)}$$

with $\gamma^{(i)} \in S$ representing the sum of all constant terms of the expression f_i and where $L(X_i)$ and $Q(X_i)$ are the set of all linear and quadratic monomials in the polynomial expression f_i, respectively.

The defining equation of X_i in the associated algebraic system over $\mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle$ looks essentially the same with an alphabet symbol $\hat{\gamma}^{(i)} \in \Sigma$ representing $\gamma^{(i)}$, i.e. $\iota(\hat{\gamma}^{(i)}) = \gamma^{(i)}$.

$$X_i = \sum_{XY \in Q(X_i)} XY + \sum_{X \in L(X_i)} X + \hat{\gamma}^{(i)}.$$

In the unfolded system the defining equations for X_i are given by

$$\begin{aligned}
X_i^{<0} &= 0\\
X_i^{=0} &= \hat{\gamma}^{(i)} + \sum_{X \in L(X_i)} X^{=0}\\
X_i^{<d} &= X_i^{<d-1} + X_i^{=d-1}\\
X_i^{=d} &= \sum_{XY \in Q(X_i)} \left(X^{=d}Y^{<d-1} + X^{<d-1}Y^{=d} + X^{=d-1}Y^{=d-1}\right) + \sum_{X \in L(X_i)} X^{=d}
\end{aligned}$$

For a polynomial expression f, a variable $X \in \mathcal{X}$ and T a variable or a semiring constant, we define the *evaluation* $f\Big|_{X=T}$ as the substitution

$$f\Big|_{X=T} := f[X \leftarrow T].$$

We extend this in the natural way to evaluate multiple variables at once: Let $X = (X_1, \dots, X_k)$ denote a vector of variables and $T = (T_1, \dots, T_k)$ be variables or semiring constants, then $f\Big|_{X=T}$ is the result of substituting each T_i for X_i.

Now we can conveniently reformulate the equation for $X^{=d}$ using derivatives:

$$\begin{aligned}
X_i^{=d} &= \sum_{Z \in \mathcal{X}} \frac{\partial f_i}{\partial Z}\Big|_{\substack{X=X^{<d-1}\\ \hat{Z}=Z^{=d}}} + \sum_{XY \in Q(X_i)} X^{=d-1}Y^{=d-1}\\
&= \sum_{Z \in \mathcal{X}} \frac{\partial f_i}{\partial Z}\Big|_{\substack{X=X^{<d-1}\\ \hat{Z}=Z^{=d}}} + \sum_{\{Y,Z\} \in \binom{\mathcal{X}}{2}} \frac{\partial}{\partial Y}\frac{\partial}{\partial Z}(f_i)\Big|_{\substack{\hat{Y}=Y^{=d-1}\\ \hat{Z}=Z^{=d-1}}}
\end{aligned}$$

Note that we have $\frac{\partial f_i}{\partial Z} = 0$ if the variable Z does not appear in the polynomial f_i.

According to the definition of the Newton approximations we have $\nu_d(X_i) = h(r_i^{(d)}) \in S$, where $r_i \in \mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle$ is the $X_i^{<d}$-component of the solution of the unfolded equations and h is the evaluation homomorphism induced by interpreting the constants Σ. Abusing notation, we let $X_i^{<d}$ in the above equations represent this value $\nu_d(X_i)$ and similarly let $X_i^{=d}$ represent the value $\Delta_d(X_i)$. Then we can express the (evaluated) unfolding as follows:

$$\begin{aligned}
\nu_0(X_i) &= 0\\
\Delta_0(X_i) &= \gamma^{(i)} + \sum_{Z \in \mathcal{X}} \frac{\partial f_i}{\partial Z}\Big|_{\substack{X=0\\ \hat{X}=\Delta_0}}\\
\nu_{d+1}(X_i) &= \nu_d(X_i) + \Delta_d(X_i)\\
\Delta_d(X_i) &= \sum_{Z \in \mathcal{X}} \frac{\partial f_i}{\partial Z}\Big|_{\substack{X=\nu_{d-1}\\ \hat{Z}=\Delta_d(Z)}} + \sum_{\{Y,Z\} \in \binom{\mathcal{X}}{2}} \frac{\partial}{\partial Y}\frac{\partial}{\partial Z}(f_i)\Big|_{\substack{\hat{Y}=\Delta_{d-1}(Y)\\ \hat{Z}=\Delta_{d-1}(Z)}}
\end{aligned}$$

We can even describe Newton's method more succinctly by writing it in vector-form. To this end, we view the evaluated Jacobian matrix $J\Big|_{X=s}$ for some point $s \in S^{\mathcal{X}}$ as a linear function $J\Big|_{X=s} : S^{[n]} \to S^{[n]}$ as follows: For $x = (x_1, \dots, x_n) \in S^{[n]}$

$$\left(J\Big|_{X=s}(x)\right)_i := \sum_{j \in [n]} J_{i,j}\Big|_{\hat{X}_j = x_j}$$

Example 3.16. *Consider the polynomial equations in 2NF*

$$\begin{aligned} X &= ZX + Yb + c \\ Y &= XZ + c \\ Z &= aX \end{aligned}$$

The Jacobian of this system assuming the variable order (X, Y, Z) *is*

$$J = \begin{pmatrix} Z\hat{X} & \hat{Y}b & \hat{Z}X \\ \hat{X}Z & 0 & X\hat{Z} \\ a\hat{X} & 0 & 0 \end{pmatrix}.$$

For $s = (u, v, w)$ *we get*

$$J\Big|_{X=s} = \begin{pmatrix} w\hat{X} & \hat{Y}b & \hat{Z}u \\ \hat{X}w & 0 & u\hat{Z} \\ a\hat{X} & 0 & 0 \end{pmatrix}$$

For $(x, y, z) \in S^3$ *we have*

$$J\Big|_{X=s}(x, y, z) = \begin{pmatrix} wx + yb + zu \\ xw + uz \\ ax \end{pmatrix} \in S^3$$

If we assume commutative multiplication, then the function $J\Big|_{X=s}$ *is given by the matrix-vector product:*

$$J\Big|_{X=s}(x, y, z) = \begin{pmatrix} w & b & u \\ w & 0 & u \\ a & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} wx + by + uz \\ wx + uz \\ ax \end{pmatrix}.$$

Setting $\nu_d = (\nu_d(X_1), \dots, \nu_d(X_n))$ and $\Delta_d = (\Delta_d(X_1), \dots, \Delta_d(X_n))$ and given the Jacobian J of the system, we get the concise vector-formulation of Newton's method for systems in C2F (or 2NF).

$$\begin{aligned}
\nu_0 &= 0 \\
\nu_{d+1} &= \nu_d + \Delta_d \\
\Delta_d &= J\Big|_{X=\nu_{d-1}}(\Delta_d) + \delta_d \\
(\delta_d)_i &= \sum_{\{Y,Z\}\in\binom{\mathcal{X}}{2}} \frac{\partial}{\partial Y}\frac{\partial}{\partial Z}(f_i)\Big|_{\substack{\hat{Y}=\Delta_{d-1}(Y)\\ \hat{Z}=\Delta_{d-1}(Z)}}
\end{aligned}$$

Note that this formulation does not yet give a closed form expression for ν_d nor does it specify how to compute Δ_d, since it is only implicitly defined as the least solution of the system of linear equations

$$\Delta_d = J\Big|_{X=\nu_{d-1}}(\Delta_d) + \delta_d.$$

Nevertheless, Δ_d is well-defined, since $J\Big|_{X=\nu_{d-1}}$ is an ω-continuous function and hence this system has a unique least solution. We discuss how to solve linear systems efficiently in Section 4.1 (for right-linear systems). Next, we describe how to obtain a closed-form description of Newton's method.

3.4.3 Commutative Semirings and Newton's Method over the Reals

In the case of commutative semirings we can give a closed form of Newton's method by representing the function $J\Big|_{X=\nu_{d-1}}$ as a matrix-vector product. Hence, the defining equations for Δ_d turn into a right-linear system of the form $X = AX + b$:

$$\Delta_d = J\Big|_{X=\nu_{d-1}} \cdot \Delta_d + \delta_d$$

It is well-known [DKV09] that the least solution to a system of right-linear equations $X = A{\cdot}X{+}b$ for an $n{\times}n$ matrix $A \in S^{[n]\times[n]}$ and $b \in S^{[n]}$ is given by $X = A^*{\cdot}b$ with $A^* := \sum_{i=0}^{\infty} A^i$. This series is called the *Neumann series* (named after Carl Neumann[2]) and is a generalization of the geometric series. It is usually studied for A being an operator over a complete normed vector space where it converges to $(I - A)^{-1}$ if $|A| < 1$.

Over semirings we cannot define the inverse of a matrix in general, but we can compute the Kleene star A^* of a matrix effectively if the Kleene star can be computed for each element (cf. Section 4.1). Applying these insights leads us to the final form of Newton's

[2]The series' name is often incorrectly attributed to John von Neumann.

method for systems in 2NF over commutative semirings:

$$\begin{aligned}\nu_{d+1} &= \nu_d + \Delta_d \\ \Delta_d &= J\Big|^{*}_{\substack{X=\nu_{d-1}\\ \hat{X}=1}} \cdot \delta_d \\ (\delta_d)_i &= \sum_{\{Y,Z\}\in\binom{\mathcal{X}}{2}} \frac{\partial}{\partial Y}\frac{\partial}{\partial Z}(f_i)\Big|_{\substack{\hat{Y}=\Delta_{d-1}(Y)\\ \hat{Z}=\Delta_{d-1}(Z)}}\end{aligned}$$

Note that the purpose of the evaluation $J\Big|_{\hat{X}=1}$ is to erase the "marked" variables $\hat{X}$ in J.

We can now show that "Newton's method" according to our definition really coincides with the well-known Newton's method over the real numbers which is commonly introduced as an approximation algorithm for finding the solution of a non-linear system of equations $G(X) = 0$ for $G : \mathbb{R}^n \to \mathbb{R}^n$.

First consider the univariate case ($n = 1$). Given a starting guess $\nu_0 \in \mathbb{R}$ for the root of G, Newton's method computes the sequence of approximations

$$\nu_{d+1} = \nu_d - \frac{G(\nu_d)}{G'(\nu_d)},$$

where G' denotes the derivative of G. Setting $G(x) = F(x) - x$ we obtain

$$\begin{aligned}\nu_{d+1} &= \nu_d - \frac{F(\nu_d)-\nu_d}{F'(\nu_d)-1} \\ &= \nu_d + \frac{1}{1-F'(\nu_d)}\cdot(F(\nu_d)-\nu_d) \\ &= \nu_d + F'(\nu_d)^* \cdot \underbrace{(F(\nu_d)-\nu_d)}_{=:\delta_d,\ (\text{see below})}\end{aligned}$$

where the last equality holds if $|F'(\nu_d)| < 1$ since then the value of the Neumann series $F'(\nu_d)^*$ is given by $\frac{1}{1-F'(\nu_d)}$.

In the multivariate case the derivative F' is replaced by the Jacobian J of F and we get

$$\nu_{d+1} = \nu_d + J\Big|^{*}_{X=\nu_d} \cdot \underbrace{(F(\nu_d)-\nu_d)}_{=:\delta_d,\ (\text{see below})}$$

It remains to justify equating δ_d with $(F(\nu_d) - \nu_d)$. To this end, we assume the perspective of derivation trees and view F as a grammar (in 2NF). From this perspective and by slightly abusing notation, $\nu_d(X)$ corresponds to the set of all derivation trees of F (with root label X) having dimension less than d. Accordingly, $\Delta_{d-1}(X)$ corresponds to the set of all trees having dimension exactly $d-1$. Hence, from the definition of δ_d by

$$(\delta_d)_i = \sum_{\{Y,Z\}\in\binom{\mathcal{X}}{2}} \frac{\partial}{\partial Y}\frac{\partial}{\partial Z}(f_i)\Big|_{\substack{\hat{Y}=\Delta_{d-1}(Y)\\ \hat{Z}=\Delta_{d-1}(Z)}}$$

we see that $(\delta_d)_i$ corresponds to the set of all derivation trees of F with root label X_i having exactly two subtrees of dimension exactly $d-1$. In particular, the trees in $(\delta_d)_i$ have dimension exactly d.

In the same way, the i-th component $(F(\nu_d) - \nu_d)_i$ corresponds to the set difference

$$D := f_i(\nu_d) \setminus \nu_d(X_i).$$

Since ν_d corresponds to the set of all trees of dimension less than d, every tree t in D has dimension exactly d. This tree t is built by combining derivation trees of dimension less than d according to the rules f_i. Since f_i is in 2NF, t must have exactly two subtrees of dimension exactly $d-1$. Hence, $D \subseteq (\delta_d)_i$ viewed as sets of trees. The other inclusion holds since $\Delta_{d-1} \subseteq \nu_d$ (viewed as sets of trees) and therefore $D = (\delta_d)_i$. By our discussion we have shown the following

Proposition 3.17. *Over the semiring* $(\mathbb{R}, +, \cdot, 0, 1)$ *of non-negative real numbers we have*

$$\delta_d = F(\nu_d) - \nu_d.$$

3.4.4 Closed Form for Non-Commutative Semirings

In Section 3.4.4 we study Newton's method over non-commutative semirings and describe how to obtain a description of the iterates via symbolic tree expressions.

For the (non-commutative) semiring of matrices with commutative entries we show a different closed form in Section 3.4.4 via the Kronecker product of matrices.

General Form via Tree Expressions

In the commutative case we can find a closed form of the Newton approximations because we can turn the linear equations defining Δ_d into a right-linear system by using commutativity of multiplication.

For non-commutative semirings we can still give a somewhat "closed-form" representation of Δ_d using regular tree expressions. We refer to the literature [CDG+07, DKV09] for a thorough treatment of tree languages and expressions, and just briefly sketch the ideas.

A *ranked* alphabet is a set of function symbols with different arity. We can denote the arity explicitly as a superscript when specifying ranked alphabets, e.g. $\Sigma = \{s^1, f^2, z^0\}$ is a ranked alphabet where s is a unary and f is a binary function symbol. Nullary symbols like "z" are also called constants. From ranked alphabets we can build *terms* in the usual way, e.g. $s(s(z))$ or $f(s(z), s(s(s(z))))$. Terms can be viewed as trees in the natural way.

Definition 3.18 (Terms). *Let Σ be a ranked alphabet. A term over Σ is defined inductively*

- *All constants $x^0 \in \Sigma$ are terms.*

- *If $f^k \in \Sigma$ is a symbol of arity k and $t_1, \dots, t_k$ are terms, then $f(t_1, \dots, t_k)$ is a term.*

Example 3.19. *We briefly describe how we can view an ordinary alphabet Σ as a ranked alphabet and words in Σ^* as terms. To this end, let $\Sigma' := \Sigma \cup \{\varepsilon\}$ with $\varepsilon \notin \Sigma$ be a ranked alphabet, where all $a \in \Sigma$ have arity 1 and ε is the only constant. Then a word $w = a_1 \cdots a_n \in \Sigma^*$ corresponds to the term $a_1(a_2(\dots a_n(\varepsilon)\dots))$. To save parentheses we can write the composition of function symbols as "$\cdot$" and we omit the "$\cdot$" when no confusion arises. Consider the word alphabet $\Sigma = \{a, b, c\}$. Then $\Sigma' = \{a^1, b^1, c^1, \varepsilon^0\}$ and $w = abaac$ corresponds to the term $t = a \cdot b \cdot a \cdot a \cdot c \cdot \varepsilon = abaac\varepsilon$. It is easy to see that we can obtain a bijection between words over Σ and terms over Σ'. Viewed as trees, the terms over Σ' are just paths ending with ε.*

Sets of terms over a ranked alphabet are called *tree languages*. Those languages can be specified in different ways, e.g. as the least solution of a system of fixpoint equations. For a function symbol $f^k \in \Sigma$, and a set of terms T we write $f(T)$ for the set of terms that can be built using f and existing terms from T, i.e. $f(T) := \{f(t_1, \dots, t_k) : t_i \in T\}$.

Example 3.20. *As a standard example we can specify the set of natural numbers using the ranked alphabet $\Sigma = \{s^1, n^0\}$ (we interpret s as the successor function and z as the value 0). The set of terms N describing the natural numbers are the least solution (w.r.t. set inclusion) of*

$$N = \{z\} \cup s(N).$$

We can also concisely describe the set of terms N via the regular tree expression $N = s(\Box_1)^{*_1} \cdot_1 z$. Here the symbol $\Box_1$ is a placeholder for substitution, the operation $\cdot_1$ denotes a substitution for the symbol $\Box_1$, and the operation "^{*_1}" denotes iterated substitution.

$$\begin{aligned} N &= \{\Box_1 \cdot_1 n\} \cup \{s(\Box_1) \cdot_1 n\} \cup \{s(s(\Box_1)) \cdot_1 n\} \cup \dots \\ &= \{n\} \cup \{s(n)\} \cup \{s(s(n))\} \cup \dots = \{n, s(n), s(s(n)), \dots\} \end{aligned}$$

Formally, we write $t[\Box_k \leftarrow u]$ for the term obtained by substituting the symbol $\Box_k$ in t by the term u. This substitution operation can be lifted to sets of terms: For sets T, U we write $T[\Box_k \leftarrow U] := \{t[\Box_k \leftarrow u] : t \in T, u \in U\}$.

Example 3.21. *If $T = \{f(\Box_1, \Box_1), g(\Box_1)\}$ and $U = \{a, b\}$ then*

$$T[\Box_1 \leftarrow U] = \{f(a, a), f(a, b), f(b, a), f(b, b), g(a), g(b)\}.$$

For a set of terms T, the set described by the expression T^{*_1} is the set of terms obtained by iterated substitution of the symbol $\Box_1$ for terms in T. More specifically,

$$\begin{aligned} T^{0_k} &:= \{\Box_k\} \\ T^{(n+1)_k} &:= T^{n_k}[\Box_k \leftarrow T] \end{aligned}$$

The Kleene closure for substitution of $\Box_k$ is then defined by taking the union over all iterates: $T^{*_k} := \bigcup_{i \in \mathbb{N}} T^{i_k}$.

Example 3.22. *Let* $T = \{f(\Box_1, \Box_1)\}$*, then*

$$T^{*_1} = \{\Box_1, f(\Box_1,\Box_1), f(f(\Box_1,\Box_1),\Box_1), f(\Box_1, f(\Box_1,\Box_1)), f(f(\Box_1,\Box_1), f(\Box_1,\Box_1)), \dots\}$$

We can solve systems of linear equations successively (similar to Gaussian elimination) and obtain regular tree expressions representing their solution. The trees represented by these expressions are built using tree substitution at a unique leaf.

Example 3.23. *Consider the system of linear equations*

$$X = aXa + bYc + e$$
$$Y = dYd + fX$$

We can first solve for Y

$$Y = (d\Box_1 d)^{*_1} \cdot_1 (fX)$$

and substitute this solution in the equation for X

$$X = aXa + b\left((d\Box_1 d)^{*_1} \cdot_1 (fX)\right)c + e$$

and hence

$$X = [a\Box_2 a + b\left((d\Box_1 d)^{*_1} \cdot_1 (f\Box_2)\right)c]^{*_2} \cdot_2 e)$$

Just like the Kleene star x^* can be seen as a shorthand for the infinite sum $\sum_{i=0}^{\infty} x^i$ we can view a regular tree expression $(x\Box_1 y)^{*_1} \cdot_1 w$ as a shorthand for the infinite sum

$$(x\Box_1 y)^{*_1} \cdot_1 w = \sum_{i=0}^{\infty} x^i w y^i$$

Over certain semirings we can transform the tree expressions representing solutions of linear equations back to standard word expressions. An example where this is possible are the (idempotent, non-commutative) lossy semirings [EKL08] describing subword-closed languages (cf. Chapter 7). In these semirings the identity $x = 1 + x$ holds and hence

$$x^n w y^n = (1+x)^n w (1+y)^n \overset{1+1=1}{=} \left(\sum_{k=0}^{n} x^k \cdot 1^{n-k}\right) w \left(\sum_{l=0}^{n} y^l \cdot 1^{n-l}\right) = \sum_{k,l=0}^{n} x^k w y^l$$

where $(1+x)^n = \sum_{k=0}^{n} x^k 1^{n-k}$ follows from idempotence and $1 \cdot x = x \cdot 1$. Again using idempotence we can then simplify the mentioned expression

$$(x\Box_1 y)^{*_1} \cdot_1 w = \sum_{n=0}^{\infty} x^n w y^n = \sum_{n=0}^{\infty} \sum_{k,l=0}^{n} x^k w y^l \overset{\text{Id.}}{=} \sum_{k,l=0}^{\infty} x^k w y^l = x^* w y^*.$$

Similarly it holds that

$$(x_0\Box_1 y_0 + \dots + x_n\Box_1 y_n)^{*_1} \cdot_1 w = (x_0 + \dots + x_n)^* w (y_0 + \dots + y_n)^*$$

This fact can be used for lossy semirings to successively transform any regular tree expression into an equivalent standard regular expression.

Example 3.24. *Consider the expression obtained in the previous example:*

$$X = [a\Box_2 a + b\,((d\Box_1 d)^{*_1} \cdot_1 (f\Box_2))\,c]^{*_2} \cdot_2 e).$$

If we assume the idempotence $x + x = x$ *and "lossiness"* $x = 1 + x$ *axioms we get*

$$X = [a\Box_2 a + b\,(d^*(f\Box_2)d^*)\,c]^{*_2} \cdot_2 e = (a+b+d+f)^* \cdot (e) \cdot (a+d+c)^*$$

Note that since $(xy)^* = ((x+1)(y+1))^* = (xy+x+y+1)^* \geqslant (x+y)^*$ *and* $(xy)^n \leqslant (x+y)^{2n}$ *for all* n*, we have* $(xy)^* = (x+y)^*$*.*

Semiring of Matrices with Commutative Entries

Natural examples of non-commutative semirings are the semirings of (square) matrices over a *commutative* semiring.

Example 3.25. *Consider the* 2×2*-matrices over the (commutative) tropical semiring over* $\mathbb{N}$*:* $(\mathbb{N}_\infty, \min, +, \infty, 0)$.

$$A := \begin{pmatrix} \infty & 1 \\ 0 & \infty \end{pmatrix} \qquad B := \begin{pmatrix} \infty & 0 \\ 1 & \infty \end{pmatrix}$$

$$A \cdot B = \begin{pmatrix} 2 & \infty \\ \infty & 0 \end{pmatrix}$$

$$B \cdot A = \begin{pmatrix} 0 & \infty \\ \infty & 2 \end{pmatrix} \neq A \cdot B$$

To show that we can effectively compute Newton approximations also for algebraic systems over matrix semirings with commutative entries, we have to show that we can obtain a closed form solution of the linear equations defining Δ_d. Over commutative semirings we have $axb = abx$ for all $a, x, b \in S$ which allows us to transform arbitrary linear equations into right-linear equations. The solution of right-linear equations can then be expressed using the Kleene star operation. We show that a similar identity holds for matrices over commutative semirings, allowing us to transform any linear system into a right-linear one.

To formulate the generalization of $axb = abx$ we use the Kronecker product of matrices [DKV09]: Given index sets I_1, J_1, I_2, J_2 we define for two matrices $A =\in S^{I_1 \times J_1}$ and $B \in S^{I_2 \times J_2}$ their *Kronecker product* (or *tensor product*) as the $(I_1 \times I_2) \times (J_1 \times J_2)$ matrix

$$(A \otimes B)_{((i_1,i_2),(j_1,j_2))} := a_{i_1,j_1} \cdot b_{i_2,j_2} \quad \text{for } i_1 \in I_1, j_1 \in J_1, i_2 \in I_2, j_2 \in J_2.$$

Observe that the matrix $(A \otimes B)$ is indexed by a tuple of tuples.

For example, for square matrices $A \in [n] \times [n]$ and $B \in [m] \times [m]$ we can view $(A \otimes B)$ as an n-by-n matrix of m-by-m sub-matrices $a_{i,j} \cdot B$:

$$(A \otimes B) = \begin{pmatrix} a_{1,1}B & \cdots & a_{1,n}B \\ \vdots & & \vdots \\ a_{n,1}B & \cdots & a_{n,n}B \end{pmatrix}$$

Note that according to our definition in Chapter 2, any matrix can be viewed as a vector over the index set of pairs $(I \times J)$, which might seem unintuitive at first but is actually very convenient. For example, given matrices $A, B, C \in S^{I \times I}$ there is only a single way we can interpret the product $(A \otimes B) \cdot C$: Since $(A \otimes B) \in S^{(I \times I) \times (I \times I)}$ it must be a matrix/vector-product and hence, C is viewed as a vector in $S^{(I \times I)}$ here. [3]

We can now state the promised generalization of $axb = abx$ for matrices. This result is known from linear algebra, where it is usually proved for matrices over $\mathbb{R}$ or $\mathbb{C}$ [Ber09]. We show that it also holds over any commutative semiring.

Lemma 3.26. *Let S be a commutative semiring and let* $A, B, X \in S^{[n] \times [n]}$ *then*

$$A \cdot X \cdot B = (A \otimes B^T) \cdot X$$

Proof. First we consider the entry (i, j) of the left-hand-side:

$$\begin{aligned}(A \cdot X \cdot B)_{i,j} &= \sum_{k=1}^{n} (A \cdot X)_{i,k} \cdot B_{k,j} \\ &= \sum_{k=1}^{n} \sum_{l=1}^{n} A_{i,l} \cdot X_{l,k} \cdot B_{k,j} \\ &= \sum_{(k,l) \in [n] \times [n]} A_{i,l} \cdot B_{k,j} \cdot X_{l,k}\end{aligned}$$

where the last equality holds since multiplication of elements in S is commutative.

Now consider the right-hand-side $(A \otimes B^T) \cdot X$ which has to be read as a matrix-vector product $M \cdot X$ with $M = (A \otimes B^T) \in S^{([n] \times [n]) \times ([n] \times [n])}$ and $X \in S^{[n] \times [n]}$. Also recall that the Matrix-Vector product between a matrix $M \in S^{I \times J}$ and a vector $X \in S^J$ is given by

$$(M \cdot X)_i = \sum_{j \in J} M_{i,j} \cdot X_j \quad \text{for } i \in I$$

[3]With this observation it becomes unnecessary to define an explicit "vectorization" function as it is common in many physics or mathematics textbooks when dealing with identities involving the Kronecker product (cf. [Ber09]).

In our case we have the index set $J = [n] \times [n]$ and thus

$$\begin{aligned}((A \otimes B^T) \cdot X)_{i,j} &= \sum_{(l,k) \in [n] \times [n]} (A \otimes B^T)_{(i,j),(l,k)} \cdot X_{l,k} \\ &= \sum_{(l,k) \in [n] \times [n]} A_{i,l} \cdot B^T_{j,k} \cdot X_{l,k} \\ &= \sum_{(l,k) \in [n] \times [n]} A_{i,l} \cdot B_{k,j} \cdot X_{l,k} \\ &= \sum_{(k,l) \in [n] \times [n]} A_{i,l} \cdot B_{k,j} \cdot X_{l,k}.\end{aligned}$$

The last equality holds since $+$ is commutative and hence the order of summation does not matter. □

Corollary 3.27. *Let* S *be a commutative semiring and let* $X = AXB + C$ *for* $A, B, C, X \in S^{[n] \times [n]}$. *Then*

$$X = (A \otimes B^T)^* \cdot C.$$

Proof. $X = AXB + C \Leftrightarrow X = (A \otimes B^T) \cdot X + C$ by the previous lemma and therefore $X = (A \otimes B^T)^* \cdot C$. □

Consider now the (non-commutative) semiring $S' = S^{[n] \times [n]}$ of square matrices over a commutative semiring S and a linear system of equations over S', i.e. given $c \in S'^{[k]}$ and a linear function $L : S'^{[k]} \to S'^{[k]}$ defined by

$$L_i(x) = \sum_{j \in [k]} a_{i,j} \cdot x_j \cdot b_{i,j}$$

we are looking for $x \in S'^{[k]}$ such that

$$x = L(x) + c.$$

By Lemma 3.26, we can transform this system of linear equations successively into a right linear system and by Corollary 3.27 we can also obtain a closed-form solution of this system. Hence, also the Newton approximations ν_d of an algebraic system can be given in closed-form.

3.5 Convergence of Newton's Method

Since the least fixpoint of an algebraic system is obtained by evaluating *all* derivation trees and since each tree has a particular dimension, we know that Newton's method converges to the least solution (in the limit). Moreover, since the dimension of a tree is at most as large as its height, we know that Newton's converges at least as fast as fixpoint iteration.

In this section we study the convergence behavior of Newton's method in greater detail. We prove bounds on the speed of convergence, and describe sufficient conditions (on the algebraic system and on the semiring) under which Newton's method reaches a fixpoint in a finite number of steps.

Our main result (Theorem 3.33) concerns the convergence speed of Newton's method over commutative semirings. An important consequence is that Newton's method converges after $n + \log \log k$ steps for n equations over a k-collapsed semirings, which generalizes the result in [EKL07b].

3.5.1 Bounds on Convergence – General Case

A very general way to describe the convergence behavior of Newton's method is to study how "fast" the Newton approximation $\mathsf{amb}_{G^{<d},X}$ converges to the true solution $\mathsf{amb}_{G,X}$. We know that the approximation converges since it eventually enumerates every derivation tree of G. Since both the approximation and the solution are power series in $\mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle$ we have that for each $w \in \Sigma^*$

$$\mathsf{amb}_{G^{<d},X}(w) \longrightarrow \mathsf{amb}_{G,X}(w) \text{ for } d \to \infty.$$

We can give a slightly stronger version of the previous statement by observing that w.l.o.g. G can be assumed to be ε-free and in C2F (and C2F preserves ambiguity): For an ε-free grammar in C2F the length of sentential forms in a derivation can not shrink and hence a derivation tree for a word w can have dimension at most $\lfloor \log|w| \rfloor$. Thus for each $w \in \Sigma^*$ we have

$$\mathsf{amb}_{G^{<\lfloor \log|w| \rfloor+1},X}(w) = \mathsf{amb}_{G,X}(w).$$

Note that in this statement, the number of Newton steps needed to guarantee that a particular coefficient of $w \in \Sigma^*$ has converged depends on $|w|$.

The best result that we can prove in the general case is for non-expansive grammars. A grammar is called non-expansive if it does not allow for a derivation $X \Rightarrow_G^* \alpha X \beta X \gamma$ for any nonterminal X. Note that we can assume w.l.o.g. that every nonterminal of G is productive and that G is ε-free. (the associated CFG of an algebraic system is always ε-free).

Examples of non-expansive grammars include all linear CFGs but also some non-linear CFGs:

Example 3.28. *The following CFG is non-expansive*

$$\begin{aligned} X &\to YZ \\ Y &\to aYb \mid \varepsilon \\ Z &\to cZd \mid \varepsilon \end{aligned}$$

The language $\mathcal{L}(X) = \{a^n b^n c^m d^m : n, m \in \mathbb{N}\}$ is not a linear context-free language which can be shown using a variant of the pumping lemma for linear CFLs (cf. [HN10]).

Note that we can check in polynomial time if a grammar is non-expansive. We show that a non-expansive grammar can only produce derivation trees of finite dimension and hence Newton's method computes the least fixpoint for non-expansive algebraic systems (defined analogously to CFGs) in a finite number of steps. Note that this proof is slightly simpler than our proof in [LS15].

Theorem 3.29 ([LS15]). *Let G be an ε-free CFG with $n := |\mathcal{X}|$ nonterminals. Newton's method computes amb in a finite number of iterations if and only if G is non-expansive.*

Moreover, if G is non-expansive then n Newton steps suffice to compute the solution, more specifically we have

$$\forall w \in \Sigma^* : \mathsf{amb}_{G,X}(w) = \mathsf{amb}_{G^{<n},X}(w).$$

Proof. First let G be expansive, witnessed by some nonterminal X (i.e. X can derive at least two copies of itself). Since every nonterminal of G can produce a word from Σ^+ we can build derivation trees $t_1, \dots, t_n$ labeled with X such that $|\mathsf{Y}(t_i)| < |\mathsf{Y}(t_{i+1})|$ and $\dim(t_i) < \dim(t_{i+1})$. Hence for each D there always exists a $w \in \Sigma^*$ with $\mathsf{amb}_{G^{<D},X}(w) < \mathsf{amb}_{G,X}(w)$ and thus Newton's method cannot compute $\mathsf{amb}_{G,X}$ in a finite number of steps.

Now assume that G is non-expansive. Consider a derivation tree t of G and recall that $\mathsf{lab}(t)$ denotes the set of labels (nonterminals from $\mathcal{X}$) appearing in t.

We prove that $\dim(t) < |\mathsf{lab}(t)|$ by structural induction on t. This proves the statement of the theorem since $|\mathsf{lab}(t)| \leqslant n = |\mathcal{X}|$ for every tree t and therefore $\mathsf{amb}_{G,X}(w) = \mathsf{amb}_{G^{<n},X}(w)$.

- If t is a leaf, then $|\mathsf{lab}(t)| = 1$ and $\dim(t) = 0$ by definition so the claim holds.
- Otherwise, assume that t has subtrees $t_1, \dots, t_m$ and the root of t is labeled with $X \in \mathcal{X}$. Since the grammar G is non-expansive, there can be at most one subtree t_i of t with $X \in \mathsf{lab}(t_i)$. Therefore, by induction we have $\dim(t_j) < |\mathsf{lab}(t)| - 1$ for $j \neq i$ and $\dim(t_i) < |\mathsf{lab}(t)|$. Hence, $\dim(t) \leqslant \max\{\dim(t_k) : k \in [m]\} < |\mathsf{lab}(t)|$.

□

Sometimes it suffices to consider non-expansive grammars. E.g. in Chapter 7 we study subword-closed languages which can be interpreted as elements of the (idempotent, non-commutative) semiring of formal languages $(2^{\Sigma^*}, \cup, \cdot, \emptyset, \{\varepsilon\})$ with the additional axiom $1 + x = x$. Such semirings are termed *lossy semirings* in [EKL08]. We show in Chapter 7 that we can transform every CFG representing a subword-closed language into an equivalent non-expansive grammar.

If the grammar G is expansive, we cannot say much about the convergence behavior of Newton's method. To illustrate that consider any expansive unambiguous grammar G, i.e. for every $w \in \Sigma^*$ we have $\mathsf{amb}_{G,X}(w) \in \{0, 1\}$. Then the least number d such that $\mathsf{amb}_{G^{<d},X}(w) = \mathsf{amb}_{G,X}(w)$ is simply the dimension of the unique derivation tree

yielding w (or 0 if $w \notin \mathcal{L}(X)$, i.e. there is no tree). Since G is expansive it possesses derivation trees of arbitrarily large dimension.

3.5.2 Bounds on Convergence – Commutative Case

Over a commutative semiring, we know that the least solution of an algebraic system is equal to the commutative ambiguity of the associated grammar. Similarly to the non-commutative case we are interested in how fast the approximation converges to the solution

$$\mathsf{camb}_{G^{<d},X} \longrightarrow \mathsf{camb}_{G,X}.$$

Using the simple observation from above in the non-commutative setting leaves us with a pointwise result: If we want to find the coefficient $\mathsf{camb}_{G,X}(w) \in \mathbb{N}_\infty$ for a *fixed* $w \in \Sigma^\oplus$, then it suffices to compute the approximate power series $\mathsf{camb}_{G^{<\log|w|+1},X}$ and extract the coefficient of w.

In the special case where $\Sigma = \{z\}$ is a unary alphabet the non-commutative and commutative setting coincide and the solution of the algebraic system is a generating function in z

$$\mathsf{camb}_{G,X} = \sum_{k=0}^{\infty} c_k z^k.$$

The Newton approximation $\mathsf{camb}_{G^{<d},X}$ is a generating function in z as well

$$\mathsf{camb}_{G^{<d},X} = \sum_{k=0}^{\infty} c'_k z^k.$$

Since we have $\mathsf{camb}_{G^{<\log|w|+1},X}(w) = \mathsf{camb}_{G,X}(w)$, the number of correct coefficients $\left|\{k \in \mathbb{N} : c'_k = c_k\}\right|$ in the approximation $\mathsf{camb}_{G^{<d},X}$ roughly doubles when increasing d to $d+1$. This fact was termed "quadratic convergence" by Pivoteau, Salvy, and Soria in [PSS12] who use Newton's method to compute approximations to generating functions for combinatorial structures.

In our convergence result, we want d not to depend on $|w|$. To get an intuition consider the following:

Remark 3.30 (Illustration of the main result)**.** *We can explain our convergence theorem via a game between two players Alice and Eve. The intuition is that Eve cannot distinguish numbers that are "too big" (e.g. not representable as a 64-bit integer). Note that in the game all information is public (i.e. players can choose their move depending on choices by the other player).*

1. *Eve chooses a number* $M \in \mathbb{N}$.
2. *Alice chooses a number* D *and computes the approximation* $\mathsf{camb}_{G^{<D},X}$.
3. *Eve chooses* $w \in \Sigma^\oplus$ *and requests the coefficient* $c = \mathsf{camb}_{G,X}(w)$ *from Alice.*

4. *Alice computes* $c' = \mathsf{camb}_{G^{<D},X}(w)$ *and reports* c' *instead of* c *to Eve.*

5. *Alice looses if* $c' \neq c$ and $c \leqslant M$ *(Eve can detect Alice' mistake).*

If Alice were to use fixpoint iteration (i.e. height-unfolding of G*) to obtain an approximation on* $\mathsf{camb}_{G,X}$ *she could never win this game since Eve could simply ask for the coefficient of some* w *which has a derivation tree of larger height than computed by Alice.*

Our result shows that Alice can win by choosing $D = n + \log\log M + 1$ *and computing the* D*-th Newton approximation* $\mathsf{camb}_{G^{<D},X}$.

In the following, let $G = (\mathcal{X}, \Sigma, \rightarrow)$ be a CFG and let $n := |\mathcal{X}|$ denote the number of nonterminals of G. Recall that we label each node of a derivation tree with the rule (X, α) used to expand the node. The first component X is called the *label* and $\mathsf{lab}(t) \subseteq \mathcal{X}$ denotes the set of all labels appearing in the tree t. We write $l(t) := |\mathsf{lab}(t)|$ to denote the number of distinct labels (i.e. nonterminals) appearing in the tree.

We call two derivation trees t, t' *Parikh-equivalent* if their root labels are the same and they yield the same word up to commutativity, i.e. $\mathsf{P}(\mathsf{Y}(t)) = \mathsf{P}(\mathsf{Y}(t'))$ with P denoting the Parikh image. In [EKL07b] it is shown that for any derivation tree containing $l(t)$ different non-terminals there exists *at least one* Parikh-equivalent tree of dimension at most $l(t)$.

Lemma 3.31 ([EKL07b])**.** *For every derivation tree* t *there is a Parikh-equivalent tree* t' *of dimension at most* $l(t)$.

Our following result extends this lemma: For every derivation tree of sufficiently large dimension there exist *many different* Parikh-equivalent trees of strictly smaller dimension.

Lemma 3.32 ([LS13, LS15])**.** *For every* $k > 0$ *and every derivation tree* t *of* G *with* $\dim(t) \geqslant l(t) + k + 1$*, there exist at least* 2^{1+2^k} *many Parikh-equivalent trees* t' *with* $\mathsf{lab}(t) = \mathsf{lab}(t')$ *and* $\dim(t') \leqslant l(t) + k$.

Proof. We proceed by structural induction on t. If t is a leaf then $\dim(t) = 0$ whereas $l(t) + k + 1 = k + 2 > 0$, so the claim trivially holds.

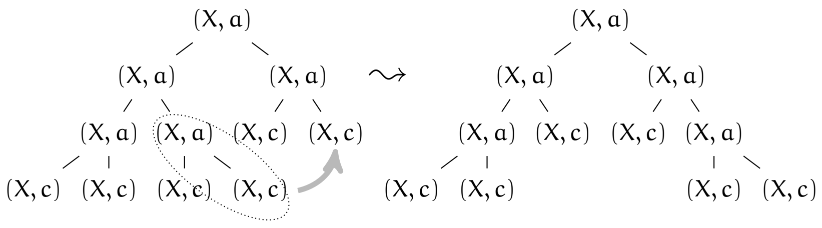
If t has a subtree s with $\dim(s) \geqslant l(t) + k + 1$ we can apply the inductive hypothesis to s, yielding at least 2^{1+2^k} Parikh-equivalent trees s' of dimension at most $\leqslant l(t) + k$ for this particular subtree. Then we apply Lemma 3.31 to every other subtree r of t to convert r to r' of dimension at most $l(t)$. Altogether, the resulting tree t' is of dimension at most $l(t) + k$ (since $k > 0$).

Therefore, we can restrict ourselves to the case where $\dim(t) = l(t) + k + 1$ and all subtrees have dimension at most $l(t) + k$. Note that in this case t must have (at least) two subtrees t_1, t_2 of dimension exactly $l(t) + k$. We distinguish two cases:

- Case $l(t_1) < l(t)$ or $l(t_2) < l(t)$: Suppose w.l.o.g. $l(t_1) < l(t)$. We can apply the inductive hypothesis to t_1, since $\dim(t_1) = l(t) + k \geqslant l(t_1) + k + 1$ and obtain at least 2^{1+2^k} Parikh-equivalent trees of dimension at most $l(t_1) + k$. Then we apply Lemma 3.31 to every *other* subtree of t producing a t' of dimension at most $l(t)+k$.

- Case $l(t_1) = l(t_2) = l(t)$. See Figure 3.3 for an illustration of this case. Since t_1 has dimension $l(t) + k$, by Lemma 3.7 t_1 contains a perfect binary tree of height $l(t)+k$ as a minor. The set of nodes of this minor on level k define 2^k (independent) subtrees of t_1. Each of these 2^k subtrees has height at least $l(t)$, and thus by the pigeonhole principle contains a path with two variables repeating. We call the partial derivation tree defined by these two repeating variables a *pump-tree*. We relocate any *subset* of these 2^k pump-trees to t_2 which is possible since $l(t_2) = l(t) = l(t_1)$ and therefore $\mathsf{lab}(t_1) = \mathsf{lab}(t_2) = \mathsf{lab}(t)$. This changes the subtrees t_1, t_2 into $\tilde{t_1}, \tilde{t_2}$.

  See the following picture for an illustration of this relocation process (here we have two possibilities to select a pump-tree from the left subtree, yielding four possible "remainders").

  Each of these 2^{2^k} choices produces a different tree $\tilde{t}$—the trees differ in the subtree $\tilde{t_1}$. As in the previous case we now apply Lemma 3.31 to every subtree of $\tilde{t}$ except $\tilde{t_1}$ to reduce the dimension of $\tilde{t}$ to at most $\dim(\tilde{t_1}) = l(t) + k$. From this we get at least 2^{2^k} different Parikh-equivalent trees of dimension at most $\dim(\tilde{t_1}) = l(t)+k$. As we can also choose t_2 as the source and t_1 as the destination of the relocation process and apply the same reasoning again, we obtain our desired lower bound of 2^{1+2^k}

□

Theorem 3.33. *For all $k > 0$ and commutative words $w \in \Sigma^\oplus$ we have that*

$$\mathsf{camb}_{G^{<n+k+1},X}(w) \geqslant \min(\mathsf{camb}_{G,X}(w), 2^{1+2^k}).$$

Proof. Let $w \in \Sigma^\oplus$ such that $\mathsf{camb}_{G^{<n+k+1},X}(w) < \mathsf{camb}_{G,X}(w)$. Therefore, there must exist some derivation tree t with $\dim(t) \geqslant n + k + 1$ and $\mathsf{P}(\mathsf{Y}(t)) = \mathsf{P}(w)$. This tree t witnesses the existence of at least 2^{1+2^k} different, but Parikh-equivalent, derivation trees yielding w up to commutativity by the previous Lemma 3.32. Hence, $\mathsf{camb}_{G^{<n+k+1},X}(w) \geqslant 2^{1+2^k}$. □

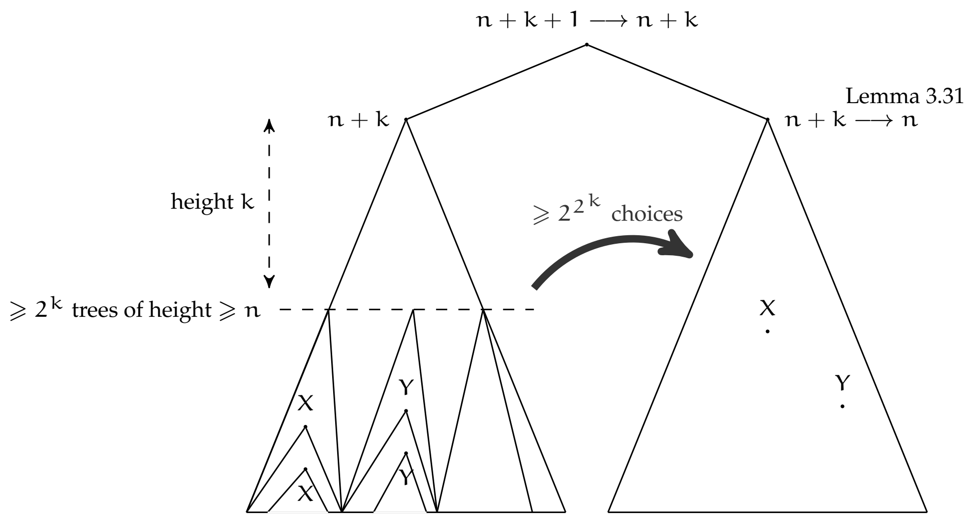


Figure 3.3: An illustration of the main case in the proof of Lemma 3.32: Let t be a binary derivation tree of dimension $n+k+1$ with two children of dimension $n+k$ such that every non-terminal occurs in the right subtree. Then t can be transformed in at least 2^{2^k} many different ways into a tree $\tilde{t}$ such that (1) $\tilde{t}$ and t have the same yield modulo commutativity and (2) $\dim(\tilde{t}) = n+k$.

Corollary 3.34. *For $k > 0$, every coefficient smaller than 2^{1+2^k} in the power series $\mathsf{camb}_{G^{<n+k+1},X}$ is correct, i.e. coincides with the coefficient of the true solution $\mathsf{camb}_{G,X}$.*

Example 3.35. *Consider the equation $X = aXX + c$ over the free commutative semiring $\mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle$. We have already seen that its solution is given by*

$$\begin{aligned} X &= \sum_{n=0}^{\infty} C_{n+1} a^n c^{n+1} \\ &= c + ac^2 + 2a^2c^3 + 5a^3c^4 + 14a^4c^5 + 42a^5c^6 + 132a^6c^7 + 429a^7c^8 + \dots. \end{aligned}$$

The dimension-unfolding of the system modulo commutativity is given by

$$\begin{aligned} X^{=0} &= c \\ X^{<d} &= X^{<d-1} + X^{=d-1} \\ X^{=d} &= 2 \cdot aX^{=d}X^{<d} + X^{=d-1}X^{=d-1} \overset{\text{comm.}}{=} (2aX^{<d})^* \cdot X^{=d-1}X^{=d-1} \end{aligned}$$

From this we can obtain the definition of the Newton iterates

$$\begin{aligned} \nu_0 &= 0 \\ \nu_{d+1} &= \nu_d + \Delta_d \\ \Delta_0 &= c \\ \Delta_d &= (2a\nu_d)^* \cdot (\Delta_{d-1})^2 . \end{aligned}$$

We report the specific values up to $d = 3$*:*

$$\begin{aligned}
\nu_0 &= 0 \\
\nu_1 &= c \\
\nu_2 &= c + (2ac)^* c^2 \\
&= c + ac^2 + 2a^2c^3 + \mathbf{4}a^3c^4 + \dots \\
\nu_3 &= c + (2a)^* c^2 + (2a(c + (2ac)^* c^2))^* \cdot (2ac)^* c^2 \cdot (2ac)^* c^2 \\
&= c + ac^2 + 2a^2c^3 + 5a^3c^4 + 14a^4c^5 + 42a^5c^6 + 132a^6c^7 + \mathbf{428}a^7c^8 + \dots \\
&\vdots \\
X &= c + ac^2 + 2a^2c^3 + 5a^3c^4 + 14a^4c^5 + 42a^5c^6 + 132a^6c^7 + 429a^7c^8 + \dots
\end{aligned}$$

In each Newton iterate we have highlighted the first coefficient which differs from the true solution, i.e. all coefficients "before" are correct. Note that the first incorrect coefficient in ν_d *is off by exactly one in this example. This is not a coincidence since* $a^{2^d-1}c^{2^d}$ *encodes some binary tree with* 2^d *leaves and all* but one *such trees have dimension less than* d *(the one with dimension* d *is the perfect binary tree of height* d*).*

To illustrate Corollary 3.34 consider $\nu_3 = \mathsf{camb}_{G^{<3},X}$*, hence we have* $k = 1$ *so the corollary tells us that every coefficient smaller than* $2^3 = 8$ *is correct. For example* $\nu_3(a^3c^4) = 5 < 8$ *is correct, as is* $\nu_3(a^{23}c^{23}) = 0 < 8$*.*

Note that this example also shows that the statement of Corollary 3.34 is far from giving tight bounds.

Consider any word $w \in \Sigma^{\oplus}$. From Lemma 3.31 we immediately obtain that $\mathsf{camb}_{G,X}(w) = 0$ if and only if $\mathsf{camb}_{G^{<n+1},X}(w) = 0$ (with $n = |\mathcal{X}|$). Put differently, if there is any derivation tree at all that yields w up to commutativity then there is one with "small" dimension. Translated to the language of algebraic systems over semirings this yields the following

Proposition 3.36 ([EKL07b])**.** *For an algebraic system* $(f_1, \dots, f_n)$ *over a commutative and idempotent semiring* S *with solution* $(s_1, \dots, s_n) \in S^{[n]}$ *we have*

$$\nu_{n+1}(X_i) = s_i.$$

Recall that a semiring is idempotent, if the identity $1 + 1 = 1$ holds (or equivalently, $x + x = x$). A generalization are semirings that satisfy the identity $1 + k = k$ for some fixed $k \in \mathbb{N}$. These are called k-collapsed semirings (cf. [BÉ09]).

Example 3.37. *The semiring of truncated natural numbers* $(\mathbb{N}_k, +, \cdot, 0, 1)$ *for* $k \geqslant 1$ *obtained by defining* $k + 1 := k$ *is* k*-collapsed. Note that the identity* $k = k + 1$ *implies* $x + k = k$ *for* $x \geqslant 0$ *and* $x \cdot k = k$ *for* $x \geqslant 1$ *by induction.*

Similarly the power series $\mathbb{N}_k\langle\langle\Sigma^\oplus\rangle\rangle$ in commuting variables with coefficients from $\mathbb{N}_k$ are a k-collapsed semiring.

A more interesting example is the k-tropical semiring $\mathcal{T}_{k,\mathbb{N}} := (\mathbb{N}_\infty^k, +, \cdot, \vec{\infty}, \vec{0})$ used to compute k-shortest paths [Moh02] (with $+, \cdot$ defined below).

For a vector $x \in \mathbb{N}_\infty^n$ we define $\min_k(x_1, \dots, x_n)$ as the sorted vector of the k smallest elements from x (w.r.t. the natural order on $\mathbb{N}_\infty$). For example $\min_3(1, 3, 5, \infty, 2, 1) = (1, 1, 2)$. If x is sorted then $\min_k$ is simply given by the first k elements of the vector. With the help of $\min_k$ we can define the addition and multiplication on $\mathcal{T}_k$ as

- $(x_1, \dots, x_k) + (y_1, \dots, y_k) := \min_k(x_1, \dots, x_k, y_1, \dots, y_k)$ *and*
- $(x_1, \dots, x_k) \cdot (y_1, \dots, y_k) := \min_k((x_i + y_j) : i, j \in [k])$ *where "+" is the usual addition on $\mathbb{N}_\infty$.*

For example we have

$$(1, 2, \infty) + (1, 4, 5) = \min_3(1, 1, 2, 4, 5, \infty) = (1, 1, 2)$$

and

$$(1, 2, \infty) \cdot (1, 4, 5) = \min_3(2, 5, 6, 3, 6, 8, \infty, \infty, \infty) = (2, 3, 5).$$

Note that for $k \geqslant 2$ the semiring $\mathcal{T}_k$ is not idempotent, e.g. consider $(1, 2) + (1, 2) = (1, 1)$ but it is k-collapsed. This follows easily from the fact that $\mathcal{T}_k$ is $(k-1)$-closed [Moh02], i.e. for all $x \in \mathcal{T}_k$

$$\sum_{i=0}^{k} x^i = \sum_{i=0}^{k-1} x^i.$$

In particular we have $1 + k = \sum_{i=0}^{k} 1 = \sum_{i=0}^{k-1} 1 = k$.

For k-collapsed semirings, we get the following result by Theorem 3.33. Note that our statement is more precise than the one we gave in [LS13, LS15].

Proposition 3.38. *Let $(f_1, \dots, f_n)$ be an algebraic system over a commutative and k-collapsed semiring S with solution $(s_1, \dots, s_n) \in S^{[n]}$. Then*

$$s_i = \begin{cases} \nu_{n+1}(X_i) & \text{if } k = 1 \\ \nu_{n+2}(X_i) & \text{if } 2 \leqslant k \leqslant 4 \\ \nu_{n+\lceil \log(\log k - 1) \rceil + 1}(X_i) & \text{if } k > 4 \end{cases}$$

Proof. To compute the solution over a k-collapsed commutative semiring, we have to make sure to discover at least k derivation trees for each word modulo commutativity (if there are that many).

From Theorem 3.33 we know $\mathsf{camb}_{G^{<n+d+1}, X}(w) \geqslant 2^{1+2^d}$, and thus it suffices to compute ν_{n+d+1} where $d \in \mathbb{N} \setminus \{0\}$ is the smallest positive number such that $k \leqslant 2^{1+2^d} \Leftrightarrow \log k - 1 \leqslant 2^d$. For $k \leqslant 4$ we get $d = 1$ and for $k > 4$ we obtain $d \geqslant \lceil \log(\log k - 1) \rceil$ as the smallest bound. □

Example 3.39. *Consider again the equation* $X = aXX + c$ *over the* k*-collapsed semirings* $\mathbb{N}_k\langle\langle\Sigma^{\oplus}\rangle\rangle$. *For* $k = 1$ *the semiring is idempotent and so we know that the solution over* $\mathbb{N}_1\langle\langle\Sigma^{\oplus}\rangle\rangle$ *is given by*

$$\nu_2 = c + (2a)^* c^2 = c + a^* c^2 = c(ac)^*.$$

For $k = 5, \ldots, 16$ *we have that* $3 \leqslant n + \lceil \log(\log k - 1) \rceil + 1 \leqslant 3$ *and hence for these* k *we need to compute* ν_3 *to correctly represent the solution. In particular we obtain the slightly stronger result that the coefficient* $\nu_3(a^4c^5) = 14$ *is correct.*

Convergence of Newton's Method over Matrix Semirings with Commutative Entries
Our results on the convergence of Newton's method over commutative semirings can also be transferred to a special case of non-commutative semirings: matrices with commutative entries. Let S be a commutative semiring. Then the semiring of matrices $S' = S^{[n]\times[n]}$ is non-commutative in general. However, an algebraic system over S' comprising k equations can simply be viewed component-wise. This component-wise system then comprises $n^2 \cdot k$ equations over S.

Hence, all our convergence results from this section also apply to the matrices with commutative entries. In particular, if S is a commutative and idempotent semiring we know that Newton's method over $S' = S^{[n]\times[n]}$ applied to an algebraic system of k equations reaches a fixpoint after at most $kn^2 + 1$ steps (similarly for commutative k-collapsed semirings S).

Example 3.40. *Consider the single algebraic equation* $X = XAX + C$ *with* $A, X, C \in S^{[2]\times[2]}$. *By viewing the equation component-wise we can expand this into an equivalent system of four algebraic equations over* S. *To this end, let*

$$X = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} \quad A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}.$$

Then we can write $X = XAX + C$ *in expanded form as*

$$\begin{aligned}
\begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} &= \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} \cdot \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot X + C \\
&= \begin{pmatrix} x_{11}a_{11} + x_{12}a_{21} & x_{11}a_{12} + x_{12}a_{22} \\ x_{21}a_{11} + x_{22}a_{21} & x_{21}a_{12} + x_{22}a_{22} \end{pmatrix} \cdot X + C. \\
&= \ldots
\end{aligned}$$

The expansion of this matrix-term results in a system of four equations defining $x_{11}, x_{12}, x_{21}, x_{22}$, *e.g.*

$$\begin{aligned}
x_{11} &= x_{11}a_{11}x_{11} + x_{12}a_{21}x_{11} + x_{11}a_{12}x_{21} + x_{12}a_{22}x_{21} + c_{11} \\
&= a_{11}x_{11}^2 + a_{21}x_{11}x_{12} + a_{12}x_{11}x_{21} + a_{22}x_{12}x_{21} + c_{11}
\end{aligned}$$

We summarize our discussion in the following proposition.

Proposition 3.41. *Let* S *be a commutative semiring. Newton's method converges*[4] *over* $S^{[k]\times[k]}$ *for all* $k \in \mathbb{N}$ *if and only if it converges over* S.

If S *is commutative and idempotent then Newton's method converges after* $n \cdot k^2 + 1$ *steps for algebraic systems over* $S^{[k]\times[k]}$.

3.6 Combinatorics of Tree Dimension

In this section we study some combinatorial parameters that are closely related to tree dimension.

We present two new contributions:

1. In Section 3.6.2 we prove a relation between tree dimension and pathwidth, answering an open question of David Eppstein.
2. In Section 3.6.3 we show how to use tree dimension to optimize the number of registers needed to evaluate arithmetic expressions.

The first was published in [LS15], the second is new.

3.6.1 Tree Dimension and Related Notions

The notion of tree dimension has been re-discovered several times under many different names during the last 60 years. We just mention a few combinatorial concepts equivalent to tree dimension (see [ELS14a] for a survey):

- The *(Horton-)Strahler number* [Hor45, Str52] is used in hydrology to classify rivers.
- The *register number* [Ers58, FRV79, Kem79] is the minimal number of registers needed to evaluate an arithmetic expression.
- A CFL is k-bounded [GS68] if it has a grammar in CNF that only generates derivation trees of dimension $k - 1$.
- The *tree-rank* of derivation trees of ET0L systems [ERV81] coincides with their dimension.
- The dimension (resp. Strahler number) is used as a hardness measure for propositional formulæ [ABLM08, GK14, BK14].

A re-occurring use of tree dimension is to describe the complexity of tree structures (see also Chapter 8). As observed by Flajolet, Raoult, and Vuillemin [FRV79], $\dim(t) + 1$

Algorithm 1: Binary tree traversal with the "cheapest subtree first" strategy.

```
Function Traverse($r(t_1, t_2)$):
    Visit ($r$)
    if $\dim(t_1) \leqslant \dim(t_2)$ then
        Traverse ($t_1$)
        Traverse ($t_2$)
    else
        Traverse ($t_2$)
        Traverse ($t_1$)
    end
    return
end
```

is the minimum amount of stack space needed by any top-down traversal of a binary tree t.

Algorithm 1 traverses a binary tree by traversing the subtree with smaller dimension first. For a binary tree t a run of the algorithm needs a stack of size $\dim(t) + 1$ to store the pending calls. On the other hand, given a perfect binary tree of height h, *any* top-down traversal has to use a stack of height $h + 1$ since the order of the subtrees visited is irrelevant.

3.6.2 Tree Dimension and Pathwidth

Many combinatorial parameters of trees are different from the dimension, but are still closely related. Some years ago, D. Eppstein asked the question whether the pathwidth of a tree is related to its Strahler number (or equivalently, its dimension) [5].

Intuitively, the pathwidth of a graph is a measure of how "path-like" the graph is: It measures the minimum amount of "abstraction" one needs to apply to the graph in order to make it a path, where "abstraction" means to form new macro-nodes containing subsets of the nodes in the graph (note that these subsets are not necessarily disjoint).

Definition 3.42 (Path decomposition)**.** *A* path decomposition *of a graph* $G = (V, E)$ *is an ordered sequence (a path)* $V_1, \ldots, V_n \subseteq V$ *such that*

1. *All vertices are covered by the sequence, i.e.* $\bigcup_{i \in [n]} V_i = V$.
2. *All edges are covered, i.e. for each* $\{u, v\} \in E$ *there is some* $j \in [n]$ *such that* $u, v \in V_j$.
3. *If* $v \in V_i \cap V_j$ *then* $v \in V_k$ *for all* k *with* $i < k < j$.

[4]By "converges" we understand convergence in finite time.
[5]See `https://en.wikipedia.org/wiki/Talk:Pathwidth` for a discussion.

Equivalently, the third condition says that for all $i \leqslant k \leqslant j$ we have $V_i \cap V_k \subseteq V_j$. Obviously every graph has a trivial path decomposition consisting just of $V_1 = V$. However, this "decomposition" is not really useful since it abstracts all $|V|$ nodes into one big set. A good measure for the granularity of a path decomposition is the notion of pathwidth:

Definition 3.43 (Pathwidth). *Let* $G = (V, E)$ *be a graph. The* width *of a path decomposition* $P = (V_1, \dots, V_n)$ *of* G *is defined as the maximum size of any* V_i *minus one,*

$$\mathrm{width}(P) := \max_{i \in [n]} |V_i| - 1.$$

The pathwidth $\mathrm{pw}(G)$ *of* G *is defined as the minimum width over all path decompositions,*

$$\mathrm{pw}(G) := \min_{\substack{P\colon \text{path dec.}\\ \text{of} G}} \mathrm{width}(P)$$

The "-1" in the definition of width is to normalize the pathwidth of any path to 1.

Proposition 3.44. *For* B_n*, the perfect binary tree* B_n *of height* n*, we have*

$$\mathrm{pw}(B_n) = \left\lceil \frac{n}{2} \right\rceil$$

Proof. We first prove $\mathrm{pw}(B_n) \leqslant \lceil \frac{n}{2} \rceil$ by induction on n. The upper bound is obviously true for $n = 0, 1$. Inductively, one can construct a path decomposition of B_n of width $w + 1$ using decompositions $(X_i)_{i \in [s]}$ of B_{n-2} of width w as follows (we identify the nodes of B_n with the strings from $\{0, 1\}^{\leqslant n}$):

$$(00X_i \cup \{0\})_{i \in [s]}, (01X_i \cup \{0\})_{i \in [s]}, \{\varepsilon, 0\}, \{\varepsilon, 1\}, (10X_i \cup \{1\})_{i \in [s]}, (11X_i \cup \{1\})_{i \in [s]}.$$

The lower bound $\mathrm{pw}(B_n) \geqslant \lceil \frac{n}{2} \rceil$ follows from a theorem on pathwidth obstructions [CDF96]. □

Different from pathwidth, the dimension is defined for rooted trees only. To make them comparable, we define $\mathrm{mindim}(T)$ as the minimum dimension of T w.r.t. the choice of the root.

Example 3.45. *For* B_n *the (rooted) perfect binary tree of height* n *we have* $\dim(B_n) = n$*. If we choose the leftmost leaf as a new root, the highest perfect binary tree contained as a minor is* B_{n-1} *(since the new root has only one child). Hence,* $\mathrm{mindim}(B_n) \leqslant n - 1$*. Further, note that* B_n *contains two* B_{n-1} *as minors and any choice of a new root leaves one of these minors intact. Thus,* $\mathrm{mindim}(B_n) = n - 1$.

We now present our main theorem on the relation of dimension and pathwidth.

Theorem 3.46. *For every tree* T *it holds that*

$$\mathrm{pw}(T) - 1 \leqslant \mathrm{mindim}(T) \leqslant 2 \cdot \mathrm{pw}(T)$$

Proof. We first show that $\mathrm{pw}(T) \leqslant 1 + \mathrm{mindim}(T)$:

Choose any $r \in V$ such that $\dim(T, r) = \mathrm{mindim}(T)$. If $\dim(T, r) = 0$, then (T, r) is by definition a chain and $\mathrm{pw}(T) \leqslant 1$ immediately follows. Thus assume $\dim(T, r) > 0$. Let $T_1, \dots, T_k$ be the connected components of $T \setminus \{r\}$, and $r_1, \dots, r_k$ the children of r in (T, r). For $(B_i^{(j)})_{i\in[s_j]}$ a path decomposition of T_j, we can construct the following path decomposition of T (to be read from left to right):

$$B_1^{(1)} \cup \{r\}, \dots, B_{s_1}^{(1)} \cup \{r\}, B_1^{(2)} \cup \{r\}, \dots, B_{s_{k-1}}^{(k-1)} \cup \{r\},\ \{r, r_k\}, B_1^{(k)}, \dots, B_{s_k}^{(k)}.$$

By definition of path decomposition, every r_j is covered by $(B_i^{(j)})_{i\in[s_j]}$, so that by adding r to the first $k-1$ decompositions all the edges $\{(r, r_1), \dots, (r, r_{k-1})\}$ are covered. After covering also the edge (r, r_k), only edges and nodes of T_k remain to be covered, so that the decomposition $(B_i^{(k)})_{i\in[s_k]}$ suffices. Hence, $\mathrm{pw}(T) \leqslant 1 + \max_{i\in[k]} \mathrm{pw}(T_i)$. In particular, if the maximum $\max_{i\in[k]} \mathrm{pw}(T_i)$ is uniquely determined, wlog. by T_k, then $\mathrm{pw}(T) \leqslant \max_{i\in[k]} \mathrm{pw}(T_i)$.

Now by induction $\mathrm{pw}(T_i) \leqslant 1 + \mathrm{mindim}(T_i) \leqslant 1 + \dim(T_i, r_i)$. By definition, either (a) $\dim(T, r) = \max_{i\in[k]} \dim(T_i, r_i)$ or (b) $\dim(T, r) = 1 + \max_{i\in[k]} \dim(T_i, r_i)$. In case (a) the maximum is unique so that for at most one subtree T_i, say T_k, we have $\mathrm{pw}(T_k) = 1 + \dim(T, r)$, while for the remaining components $\mathrm{pw}(T_i) < 1 + \dim(T, r)$. Thus $\mathrm{pw}(T) \leqslant 1 + \dim(T, r) = 1 + \mathrm{mindim}(T)$ by choice of r. In case (b) we have $\mathrm{pw}(T_i) \leqslant 1 + \dim(T_i, r_i) \leqslant \dim(T, r)$, and thus $\mathrm{pw}(T) \leqslant 1 + \dim(T, r)$, too.

It remains to show that $\mathrm{mindim}(T) \leqslant 2\mathrm{pw}(t)$ (the proof is similar to the proof of Lemma 9 in [Sud04]):

Set $w := \mathrm{pw}(t)$. If $w = 0$, then t consists of a single node and $\mathrm{mindim}(T) = 0$.

Thus assume $w > 0$ and let $(B_i)_{i\in[s]}$ be some path decomposition of minimal width w. Choose any $v_1 \in B_1$ and $v_s \in B_s$, and let π be the unique simple path connecting v_1, v_s in T. As before, let $r_1, \dots, r_k$ be the nodes of T which are not located on π but connected to some node along π by an edge; then denote by (T_i, r_i) the corresponding subtrees w.r.t. (T, v_1). Since every edge of π is covered by at least one B_i $(B_i - \{v_1, \dots, v_s\})_{i\in[s]}$ is a path decomposition of width at most $w - 1$ for every tree T_i, i.e., $\mathrm{pw}(T_i) \leqslant w - 1$. By induction, we therefore have

$$\dim(T_i, r_i) \leqslant 1 + \mathrm{mindim}(T_i) \leqslant 1 + 2\mathrm{pw}(T_i) \leqslant 1 + 2(w - 1).$$

Thus

$$\mathrm{mindim}(T) \leqslant \dim(T, v_1) \leqslant 1 + \max_{i\in[s]} \dim(T_i, r_i) \leqslant 2w = 2\mathrm{pw}(T). \qquad \square$$

Note that the bounds in Theorem 3.46 are sharp up to ± 1: For the left inequality consider the perfect ternary tree T_n of height n: No matter which node we pick as root the resulting rooted tree contains two T_{n-1} minors so that induction immediately yields that $\mathrm{mindim}(T_n) = n$. On the other hand we have $\mathrm{pw}(T_n) = n$ (see e.g. [Sud04]).

Similarly for the right inequality we use B_n the perfect binary tree B_n of height n. Here we have $\mathrm{mindim}(B_n) = n - 1$ and $\mathrm{pw}(B_n) = \frac{n}{2}$: Our theorem yields the lower bound $\mathrm{pw}(B_n) \geqslant \frac{n-1}{2} = \lfloor \frac{n}{2} \rfloor$, so it is off by 1 for perfect binary trees of odd height.

3.6.3 Application: Optimizing Arithmetic Expressions

The "cheapest subtree first" strategy used by Algorithm 1 to traverse binary trees can also be used to evaluate arithmetic expressions given as a binary syntax tree in a register-optimal way as first observed by Ershov [Ers58, FRV79].

Ershov only considers expressions given as a fixed syntax tree. However, using associativity we can write arithmetic expressions in syntactically different but semantically equivalent ways. [6]

Example 3.47. *The syntax tree of the expression* $(a + b) + (c + d)$ *has dimension* 2*. The equivalent expression* $(a + (b + (c + d)))$ *has a syntax tree of dimension* 1*. Hence, evaluation of the expression requires* 3 *registers in its first but only* 2 *in its second form.*

Formally, we consider the following problem of optimizing expressions modulo associativity.

Problem 3.48 (MINREGASSOC)**.** *Given an arithmetic expression* e *(as a binary syntax tree), find an expression* e' *equivalent to* e *modulo associativity of multiplication and addition such that the number of registers needed to evaluate* e' *is minimal.*

In the previous example the optimal bracketing of the expression was given by a right-most bracketing of the "flattened" expression $(a + b + c + d)$ which corresponds to a right-most binarization of the associated 4-ary syntax tree.

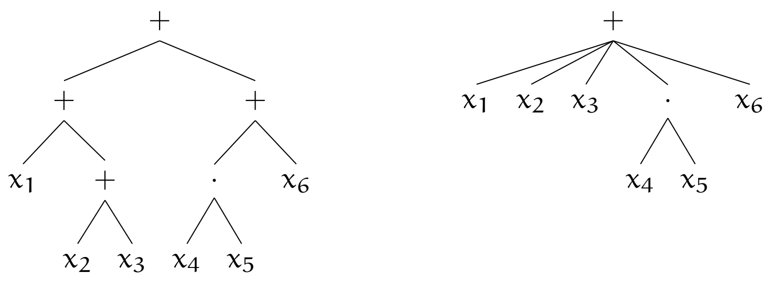
Combinatorially, a *binarization* of a tree t (of arbitrary arity) is a binary tree t' obtained from t by contracting inner nodes of degree 2 and by introducing new inner nodes such that t' has the same leaves as t in the same left-to-right order. For example, binarizing a derivation tree of a context-free grammar is analogous to transforming a grammar into CNF. [7]

It is easy to see that the MINREGASSOC problem can be reduced to finding an optimal binarization of the associated "flattened" syntax tree. The following example illustrates the operation of flattening a tree by collapsing nodes of the same type. Note that we do not assume commutativity of the operations here.

Example 3.49. *Here we see a binary syntax tree for an arithmetic operation on the left and its flattened version on the right. Note that the tree on the left has dimension* 2 *while the flattened version has dimension* 1*.*

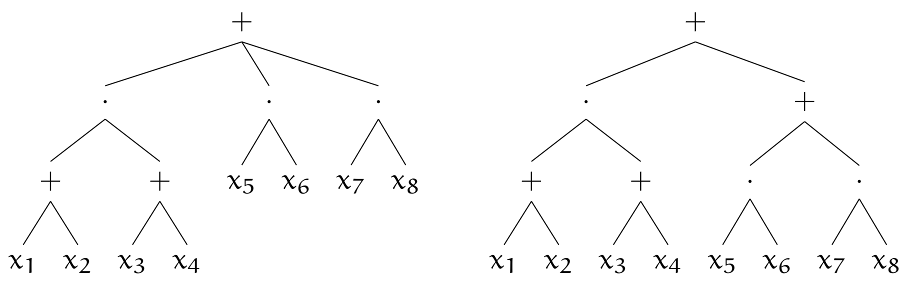
[6] We do not consider commutativity of either addition or multiplication here, so we can view expressions as terms modulo associativity.

[7] In the case of arithmetic expressions, contraction of unary "minus" nodes does not change the number of registers needed for evaluation but changes the semantics. However, we can recover a semantically equivalent expression by reverting the contractions in the binarized tree.

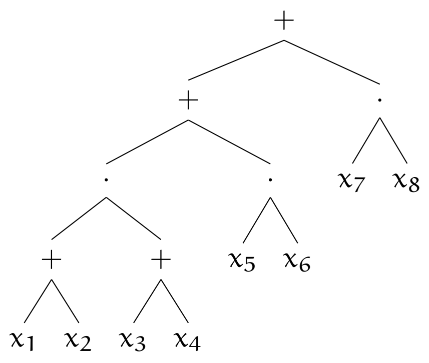


For the flat expression $(a + b + c + d)$ one optimal binarization can be obtained by a right-most binarization. However, such a right-most binarization of the flattened tree need not be optimal:

Example 3.50. *Consider the ternary expression tree on the left of dimension* 2*. A right-most binarization gives the binary tree on the right of dimension* 3*.*



A different binarization gives a tree of dimension 2*:*



In this fashion we can construct more examples where a simple left-most or right-most binarization strategy gives a suboptimal tree (i.e. a tree with higher dimension than the flattened tree). Note that for a suboptimal syntax tree *any* evaluation strategy needs more registers than actually necessary (given an optimal tree).

However, we show next in Theorem 3.51 that for any tree t (of arbitrary arity) there always exists a binarization with the same dimension as t. Hence for any expression we can find a binary syntax tree with the same dimension as its flattened tree. Furthermore note that the dimension of the flattened tree is a lower bound on the dimension of *any* binarization.

Putting these results together we can solve the MINREGASSOC problem by flattening the syntax tree of the input expression and then building an optimal binarization via Theorem 3.51.

Theorem 3.51. *For every tree* t *there is a binarization* t' *such that* $\dim(t) = \dim(t')$.

Proof. First note that by removing unary nodes (nodes with degree 2) from the tree we can assume that every inner node has degree at least 2.

We proceed by induction on the number of nodes $n := |V(t)|$. For $n = 1$, t only consists of a leaf and already is a binary tree.

Now let $n > 1$, $D := \dim(t)$, and let $t_1, \dots, t_k$ be the subtrees of t ordered from left to right. If $k = 2$ we can binarize each subtree inductively which results in a binary tree t'. Hence, in the following we assume $k \geqslant 3$. Since $\dim(t) = D$ one of the following cases must be true:

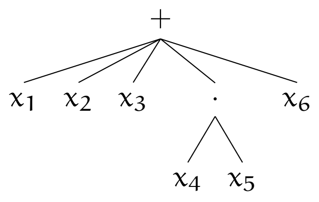
1) t has (exactly) one subtree t_i of dimension D and all other subtrees have dimension at most $D - 1$: Assume w.l.o.g. that $i > 1$, i.e. t_i is not the left-most tree (otherwise we argue symmetrically). We form two new trees, $s_1 = t_1$ and s_2 a new node with subtrees $t_2, \dots, t_k$. Since s_1 and s_2 have fewer nodes than t we can binarize s_1 and s_2 by induction yielding binary trees s'_1 and s'_2 with $\dim(s'_1) \leqslant D - 1$ and $\dim(s'_2) = D$. Hence, we can build the binarization t' of t by attaching the two subtrees s'_1, s'_2 to a new root and $\dim(t') = D$.

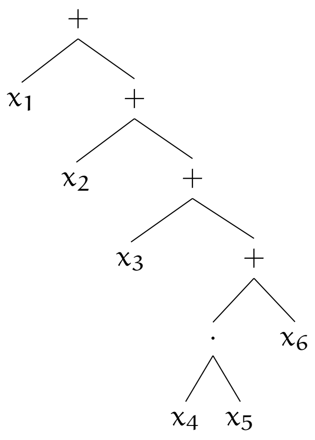
2) t has at least two subtrees t_i and t_j of dimension $D-1$ (and all other subtrees have dimension at most $D - 1$): We form two new trees, s_1 with subtrees $t_1, \dots, t_i$ and s_2 with subtrees $t_{i+1}, \dots, t_k$[8]. Note that $\dim(s_1) = D - 1 = \dim(s_2)$ and they both have fewer nodes than t so again by induction they can be binarized yielding binary trees s'_1, s'_2 with $\dim(s'_1) = D - 1 = \dim(s'_2)$. Attaching s'_1 and s'_2 to a new root, yields the binarization t' of t with $\dim(t') = D$.

□

Example 3.52. *Consider the flattened tree on the left from the example above. Applying the binarization procedure from the proof of Theorem 3.51 yields the tree on the right of dimension* 1.

[8]If $i = 1$ we do not add a new root, so $s_1 = t_1$, similarly for $i + 1 = k$ and s_2.





Corollary 3.53. *The* MINREGASSOC *problem can be solved in polynomial time.*

3.7 Conclusions

In this chapter we have extended the theoretical foundations of Newton's method over semirings. Specifically, we made the following contributions:

- We have shown how obtain a closed form of Newton's method over non-commutative semirings and in particular over matrix semirings.
- We have proved a general convergence theorem for Newton's method over commutative semirings which allows us to obtain that Newton's method converges in finite time over all k-collapsed semirings.
- We have proved a tight relation between tree dimension and pathwidth and shown how tree dimension can be used to optimize arithmetic expressions w.r.t. the number of registers needed to evaluate them.

4 Newton's Method: Generic Algorithms and Implementation

As we have seen, Newton's method is a general procedure to approximate or even solve algebraic systems over special semirings (e.g. k-collapsed semirings). So far, we have formulated the ideas in a theoretical and non-algorithmic way.

In this chapter we first describe the algorithmic details of Newton's method in Section 4.1. In particular, we provide a detailed complexity analysis of different methods for solving right-linear equations – a problem which is central to Newton' method over commutative semirings. Then, in Section 4.2 we present an overview of our generic library FPSOLVE implementing these algorithms.

This chapter is based on work that has appeared in [STL13, ELS14b, ELS15].

4.1 Algorithmic Details of Newton's Method

Here we describe the main algorithms needed to implement Newton's method and discuss some optimizations over special semirings. In Section 4.1.1 we give the general formulation of Newton's method and discuss more efficient versions for idempotent and numeric semirings. Next, in Section 4.1.2 we describe how Newton's method We compare several methods to solve right-linear systems and provide a detailed complexity analysis in Section 4.1.3. Finally, we briefly describe two general algorithmic optimization techniques: Symbolic solving (Section 4.1.4), and SCC decomposition (Section 4.1.5).

In the following we restrict ourselves to algebraic systems in 2NF which is (1) no real restriction as described before, (2) makes the presentation easier, and (3) leads to more efficient algorithms.

4.1.1 Generic Formulation of Newton's Method

As we have seen in Section 3.4, in general we can define Newton's method (for systems in 2NF) as follows:

$$\begin{aligned} \nu_0 &= 0 \\ \nu_{d+1} &= \nu_d + \Delta_d \\ \Delta_d &= J\Big|_{X=\nu_{d-1}}(\Delta_d) + \delta_d \\ (\delta_d)_i &= \sum_{\{Y,Z\}\in\binom{\mathcal{X}}{2}} \frac{\partial}{\partial Y}\frac{\partial}{\partial Z}(f_i)\Big|_{\substack{\hat{Y}=\Delta_{d-1}(Y)\\ \hat{Z}=\Delta_{d-1}(Z)}} \end{aligned}$$

This suggests the general form of Newton's method described in Algorithm 2.

Algorithm 2: Generic algorithm for computing the Newton sequence.

```
input  : number of iterations $N$, vector $F$ of polynomials, vector of variables $\mathcal{X}$
output: vector of semiring values $\nu$ (the N-th Newton approximation of the
        solution to $\mathcal{X} = F(\mathcal{X})$)
InitLinearSolver($F, \mathcal{X}$) ;                /* Computes the Jacobian $J$ */
$\delta := F(\vec{0})$
$\nu := \vec{0}$
$\Delta := \vec{0}$
for $d = 1 \dots N$ do
    $\Delta :=$ SolveLinear($\nu, \delta$) ;                    /* Computes $\Delta_d$ */
    $\delta :=$ GenerateDelta($\Delta, F$) ;                   /* Computes $\delta_d$ */
    $\nu := \nu + \Delta$ ;                                    /* Computes $\nu_d$ */
end
return $\nu$
```

To instantiate this generic algorithm we have to specify how to implement the functions `InitLinearSolver`, `SolveLinear`, and `GenerateDelta`. The function `GenerateDelta` computes the value δ_d and the function `SolveLinear` solves the linear system

$$\Delta_d = J\Big|_{X=\nu_{d-1}}(\Delta_d) + \delta_d$$

in every iteration. The implementation of `InitLinearSolver` depends on whether we use the symbolic method of solving linear systems (cf. Section 4.1.4 below), but in any case this function computes the Jacobian of F and stores it so `SolveLinear` can use it.

The definition of Newton's method as unfolding (cf. 3.4) also suggests a general way to define the `GenerateDelta` function (see Algorithm 3).

Algorithm 3: Generic delta-computation.

```
input  : vector of semiring values $\Delta$, vector $F$ of $n$ polynomials of degree $\leqslant 2$
output: vector of semiring values $\delta$
$\nu := \vec{0}$
$\Delta := \vec{0}$
for $i = 1 \ldots n$ do
    for $\{X, Y\} \in \binom{\mathcal{X}}{2}$ do
        $\delta_i := \delta_i + \frac{\partial}{\partial X}\frac{\partial}{\partial Y} F_i \Big|_{\substack{\hat{X}=\Delta(X) \\ \hat{Y}=\Delta(Y)}}$
    end
end
return $\delta$
```

Note that it was shown in [EKL10] that δ_d can be chosen as any vector satisfying $\nu_d + \delta_d = F(\nu_d)$. Such a δ_d always exists since F is monotone and furthermore the sequence ν_d is independent of the choices of δ_d. Hence, the generic algorithm to compute δ_d derived from the unfolding does not constitute the only way to obtain a suitable δ_d. In fact if the semiring is idempotent or admits a well-defined subtraction operation there are more efficient means to compute δ_d.

Idempotent Semirings Over idempotent semirings Newton's method can be simplified in two ways (cf. [EKL07b, EKL10]): First, δ_d can be set to $\delta_d = F(0)$ in every iteration and second, we can save one addition due to idempotence by setting $\nu_d = \Delta_{d-1}$ (see [EKL10]). Hence δ_d can be computed once (in linear time) and then be re-used in every iteration to compute Δ_d by solving a linear system (which already computes the next ν_d).

Numeric Semirings We call a semiring *numeric* if we can define and compute the modified subtraction operation: S is a numeric semiring if for $x, y \in S$ with $x \leqslant y$ there is exactly one $c \in S$ such that $x+c = y$, with a computable c. Hence, for numeric semirings we can then define the difference $y - x := c$.

In numeric semirings, we can speed up the computation of the Newton approximations by specializing the `GenerateDelta` function: Since $\nu_d \leqslant F(\nu_d)$ we can simply compute $\delta_d = F(\nu_d) - \nu_d$ as explained in Section 3.4.3.

Algorithm 4: Newton's method over idempotent semirings.

```
input  : number of iterations $N$, vector $F$ of polynomials, vector of variables $\mathcal{X}$
output: vector of semiring values $\nu$, the $N$-th Newton approximation of the
        solution to $\mathcal{X} = F(\mathcal{X})$

InitLinearSolver($F, \mathcal{X}$) ;           /* Computes the Jacobian $J$ */
$\delta := F(\vec{0})$
$\nu := \vec{0}$
$\Delta := \vec{0}$
for $d = 1 \ldots N$ do
    $\nu :=$ SolveLinear($\nu, \delta$) ;                   /* Computes $\nu_d$ */
end
return $\nu$
```

Examples of numeric semirings are all rings like the real numbers $\mathbb{R}$ with ordinary addition and multiplication or the natural numbers $\mathbb{N}$. Strictly speaking, $\mathbb{R}_\infty^{\geqslant 0}$ (the ω-continuous extension of $\mathbb{R}^{\geqslant 0}$) is not a numeric semiring since we cannot define subtraction for the element ∞. However, for all other elements $x \leqslant y$ we can define the subtraction and thus we can also speed up the computation of Newton's method over $\mathbb{R}_\infty^{\geqslant 0}$.

4.1.2 Specialization for Commutative Semirings

If the underlying semiring is commutative we can find more specialized and efficient algorithms for computing δ_d and for solving linear equations. As a first step we can exploit commutativity to compute derivatives of polynomials more efficiently. First recall that for $n \in \mathbb{N}, s \in S$ we write $n \cdot s := \sum_{i=1}^{n} s$, which can be computed using $\mathcal{O}(\log n)$ additions (analogous to the square-and-multiply method for fast exponentiation). Now consider as an example the polynomial $f = aX^2Y + bX^3$ then $\frac{\partial f}{\partial X} = 2aXY + 3bX^2$ and $\frac{\partial f}{\partial X^2} = 2aY + 6bX$. In general, let $X = (X_1, \ldots, X_n)$ and let p be a commutative polynomial which we can write concisely in multi-index notation as

$$p = \sum_{i \in \mathbb{N}^{[n]}} a_i X^i = \sum_{i \in \mathbb{N}^{[n]}} a_i X_1^{i_1} \cdot X_n^{i_n}$$

with $a_i = 0$ for all but finitely many i. For a multi-index $d = (d_1, \ldots, d_n) \in \mathbb{N}^{[n]}$ the derivative $\frac{\partial p}{\partial X^d}$ can be written as

$$\frac{\partial p}{\partial X^d} = \sum_{i \in \mathbb{N}^{[n]}} i^{\underline{d}} a_i' X^{i-d}.$$

where $a'_i = \begin{cases} 0, & \text{if } i - d < 0 \\ a_i, & \text{otherwise} \end{cases}$ and the generalized falling factorial $i^{\underline{d}}$ is defined as $i^{\underline{d}} := i_1^{\underline{d_1}} \cdot \dots \cdot i_n^{\underline{d_n}}$.

Example 4.1. *Let* $\vec{X} = (X, Y, Z)$ *and* $d = (1, 0, 2)$ *then for* $p = aX^3Z^2 + bY^3 + cX^3Z + dX^4YZ^4$ *we have*

$$\frac{\partial p}{\partial \vec{X}^d} = 3 \cdot 2 \cdot 1 \cdot aX^2Z^0 + 0 + 0 + 4 \cdot 4 \cdot 3 \cdot dX^3YZ^2 = 6aX^2 + 48dX^3YZ^2$$

As a result, we can compute derivatives (for the Jacobian matrix and the δ_d) in linear time by considering each monomial once (assuming polynomials of degree at most 2).

As seen in Section 3.4.3, over a commutative semiring, the system of linear equations defining Δ_i turns into a right-linear system of the form $x = Ax + b$ whose least solution is given by A^*b. For the linear system defining Δ_d in Newton's method this yields

$$\Delta_d = J\Big|^{*}_{\substack{X=\nu_{d-1} \\ \hat{X}=1}} \cdot \delta_d$$

There are many choices how to implement the `SolveLinear` function of our generic algorithm to compute Δ_d. We have two possibilities to compute the matrix $J\Big|^{*}_{\substack{X=\nu_{d-1} \\ \hat{X}=1}}$:

1. Precompute J^* as a matrix of symbolic expressions in variables $\mathcal{X}$. In each iteration evaluate this symbolic matrix at the point $X = \nu_{d-1}$.

2. In each iteration, first evaluate the Jacobian J at $X = \nu_{d-1}$ and then compute J^* over the semiring S.

These two choices give rise to two different solving algorithms which we term *symbolic* and *concrete* solving, respectively (cf. Section 4.1.4).

Remark 4.2. *The third possibility to compute* Δ_d *is to avoid computing the starred Jacobian* J^* *altogether. For comparison, consider the problem of solving linear equations encountered in numerical linear algebra. Given an invertible matrix* $A \in \mathbb{R}^{n \times n}$ *and* $b \in \mathbb{R}^n$ *we look for* $x \in \mathbb{R}^n$ *such that* $A \cdot x = b$*. Although the solution formally is given by* $x = A^{-1} \cdot b$*, this equation should not be read literally as an algorithm to compute* x *because: (1) matrix inversion is a badly conditioned problem in general, (2) even if* A *is sparse, its inverse* A^{-1} *can be very dense, and (3) if* A *has a special structure (e.g.* A *is a band or a Toeplitz matrix) there are more efficient algorithms to solve the system. The paper by Litvinov et al. [LRSS13] surveys "universal" algorithms that solve such linear equations over semirings.*

4.1.3 Solving Right-Linear Systems

Here, we compare several methods to solve systems of (right-)linear equations $x = Ax + b$ over a semiring S, where $x \in S^{[n]}, A \in S^{[n]\times[n]}$. Such systems are studied under the name *Bellman equation* [LRSS13] in tropical and idempotent mathematics. The least solution of such a linear system over an ω-continuous semiring is given by A^*b where $A^* = \sum_{n\geqslant 0} A^n$. It is well-known that A^* can be effectively computed over all semirings for which we can compute the Kleene star of elements [DKV09]. Hence, solving linear systems can be reduced to computing the Kleene star of matrices. However, as we remarked above this might not be the most efficient solution as illustrated by matrices over the real semiring $(\mathbb{R}^{\geqslant 0}_{\infty}, +, \cdot, 0, 1)$: If $\|A\| < 1$ we have $A^* = (I - A)^{-1}$, so the Kleene star of a matrix corresponds to inverting $(I - A)$ which is a badly conditioned problem in general. So over numeric domains we should avoid computing A^* and solve the linear system directly, e.g. via a generalizations of the well-known LU-decomposition.

General Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is a well-known procedure to solve the all-pairs-shortest-path problem in weighted directed graphs: Given the weighted adjacency matrix A of a digraph with entries in $\mathbb{R}^{\geqslant 0}_{\infty}$, the entry (i, j) of the matrix $A^* = \sum_{i=0}^{\infty} A^i$ gives the weight of the shortest path from node i to node j. In this special case, sum, product, and Kleene star are interpreted over the real tropical semiring $(\mathbb{R}^{\geqslant 0}_{\infty}, \min, +, \infty, 0)$ and we have in fact that $A^* = \sum_{i=0}^{n-1} A^i$ for an $n \times n$ matrix A.

The Floyd-Warshall algorithm can be stated in a very general form for an arbitrary semiring (with computable Kleene star) [DKV09]. In Algorithm 5 we describe a slightly optimized version from [DKV09] which avoids the re-computation of certain expressions.

From the description of the algorithm we can easily count the number of semiring operations needed. The algorithm needs n Kleene star computations, $n^3 - n$ multiplications, and $n^3 - 2n^2 + n$ additions, so the total number of operations needed is $2n^3 - 2n^2 + n \in \Theta(n^3)$.

Divide-And-Conquer Algorithm

The Kleene star of a matrix can be defined recursively (cf. [DKV09]) We assume a subdivision of the matrix $M \in S^{[n]\times[n]}$ into four matrices $A \in S^{[n_1]\times[m_1]}$, $B \in S^{[n_1]\times[m_2]}$, $C \in S^{[n_2]\times[m_1]}$, and $D \in S^{[n_2]\times[m_2]}$, such that $n_1 + n_2 = n$ and $m_1 + m_2 = n$.

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

Algorithm 5: Generalized Floyd-Warshall algorithm over semirings.

```
input  : Matrix $A \in S^{n \times n}$ over a semiring $S$.
output: Reflexive-transitive closure $A^*$.

$B := A$
for $k = 1 \dots n$ do
    $B_{k,k} := B^*_{k,k}$
    for $i = 1 \dots n, i \neq k$ do
        $B_{i,k} := B_{i,k} \cdot B_{k,k}$
        for $j = 1 \dots n, j \neq k$ do
            $B_{i,j} := B_{i,j} + B_{i,k} \cdot B_{k,j}$
        end
    end
    for $j = 1 \dots n, j \neq k$ do
        $B_{k,j} := B_{k,k} \cdot B_{k,j}$
    end
end
return B
```

Then we obtain

$$M^* = \begin{bmatrix} F & \gamma \\ G^*\beta & G^* \end{bmatrix} \quad \text{with} \quad \begin{aligned} \alpha &= A^*B \\ \beta &= CA^* \\ G &= D + C\alpha \\ \gamma &= \alpha G^* \\ F &= A^* + \gamma\beta \end{aligned}$$

An implementation in pseudo-code of this method is shown in Algorithm 6 (note that we do not need to store the terms β and γ by using the already computed blocks of the output-matrix).

Remark 4.3. *As remarked in [LRSS13], this algorithm can also be viewed as the escalator method for matrix inversion known from numerical linear algebra.*

This algorithm performs two recursive calls (to compute A^* and G^*), two matrix additions, and six matrix multiplications. If we assume a division of the matrix into equal sized submatrices, the number of operations needed by this algorithm can be expressed via the following recurrence relation

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + 6\left[2\left(\frac{n}{2}\right)^3 - \left(\frac{n}{2}\right)^2\right] + 2\left(\frac{n}{2}\right)^2 \\ &= 2T\left(\frac{n}{2}\right) + \frac{3}{2}n^3 - n^2 \end{aligned}$$

Algorithm 6: Divide-and-conquer computation of the Kleene star of a matrix.

```
input  : Matrix $A \in S^{n \times n}$ over a semiring $S$.
output: Reflexive-transitive closure $A^*$.

Procedure Kleene-Star($A$):
    if $A \in S^{[1] \times [1]}$ then
        return $A^*$;
    end
    $A := A_{1:\lfloor n/2 \rfloor, 1:\lfloor n/2 \rfloor}$
    $B := A_{1:\lfloor n/2 \rfloor, \lceil n/2 \rceil:n}$
    $C := A_{\lceil n/2 \rceil:n, 1:\lfloor n/2 \rfloor}$
    $D := A_{\lceil n/2 \rceil:n, \lceil n/2 \rceil:n}$
    $S :=$ Kleene-Star($A$)
    $\alpha := S \cdot B$
    $M_4 :=$ Kleene-Star($D + C \cdot \alpha$)
    $M_3 := M_4 \cdot C \cdot S$
    $M_2 := \alpha \cdot M_4$
    $M_1 := \alpha \cdot M_3 + S$
    return $\begin{pmatrix} M_1 & M_2 \\ M_3 & M_4 \end{pmatrix}$;
end
```

For $T(n) = 1$ and $n = 2^l$ we can obtain by induction that $T(n) = 2n^3 - 2n^2 + n \in \Theta(n^3)$. We can see that this algorithm uses the same number of operations as Floyd-Warshall: More specifically, both algorithms need $n^3 - 2n^2 + n$ additions, $n^3 - n$ multiplications, and n Kleene stars.

In our analysis we assumed that the schoolbook method is used to multiply two $n \times n$ matrices which needs n^3 multiplications and $n^3 - n^2$ additions. Since we do not have a difference operator, we cannot use more sophisticated methods like Strassen's algorithm. In fact, Kerr showed a lower bound of $\Omega(n^3)$ operations for generic matrix multiplication over algebraic structures without an additive inverse (e.g. semirings[1]) [Ker70]. Note that this result does not preclude the existence of faster algorithms for special semirings. For example over the tropical semiring there exist subcubic algorithms (cf. [Zwi02]).

[1]More precisely, Kerr showed the lower bound for the semiring of natural numbers $(\mathbb{N}, +, \cdot, 0, 1)$ which implies the general result.

LDU-decomposition

If the matrix A in the system $x = Ax + b$ is a lower or upper triangular matrix, we can solve the system easily by forward or backward substitution, respectively. This is analogous to the LU decomposition from numerical linear algebra.

In [LRSS13] it is shown how to obtain from A three matrices (L, D, U) where L is lower triangular, U is upper triangular, and D is a diagonal matrix, such that $A^* = U^*D^*L^*$. Such a triple (L, D, U) is called an *LDU-decomposition* of A and can be computed in $\Theta(n^3)$ operations as well as we show below[2]. If we have an LDU-decomposition of A we can solve the system $x = Ax + b$ by solving the three simple systems

$$z = Lz + b \quad y = Dy + z \quad x = Ux + y.$$

Proposition 4.4. *Let* $L \in S^{[n]\times[n]}$ *with* $L_{i,j} = 0$ *for* $i \leqslant j$ *be a strictly lower triangular matrix,* $D \in S^{[n]\times[n]}$ *with* $L_{i,j} = 0$ *for* $i \neq j$ *be a diagonal matrix, and* $U \in S^{[n]\times[n]}$ *with* $U_{i,j} = 0$ *for* $j \leqslant i$ *be a strictly upper triangular matrix. Then*

$$\begin{aligned} D^* &= \mathrm{diag}(D^*_{1,1}, \ldots, D^*_{n,n}) \\ L^* &= \sum_{i=0}^{n-1} L^i \\ U^* &= \sum_{i=0}^{n-1} U^i \end{aligned}$$

Proof. By induction on i we obtain $D^i = \mathrm{diag}(D^i_{1,1}, \ldots, D^i_{n,n})$ and hence the statement for D follows. Strictly triangular matrices such as L and U are nilpotent of degree n, i.e. $L^n = U^n = 0$. It follows that $L^* = \sum_{0\leqslant i\leqslant n-1} L^i$ and $U^* = \sum_{0\leqslant i\leqslant n-1} U^i$. □

The systems involving the lower and upper triangular matrices L and U can be efficiently solved by forward and backward substitution, respectively. From the Algorithms 7 and 8 we see that these require $\frac{n(n-1)}{2}$ additions and multiplications each. Solving the system $y = Dy + z$ requires n Kleene stars and n multiplications.

To derive the LDU decomposition, consider the matrix M subdivided into

$$M = \begin{bmatrix} a & b \\ c & D \end{bmatrix}$$

with $a \in S^{[1]\times[1]}$, $b \in S^{[1]\times[n]}$, $c \in S^{[n]\times[1]}$, and $D \in S^{[n-1]\times[n-1]}$. M can be regarded as the weighted adjacency matrix of a directed graph. If we define the weight of a path in this graph as the product of the weights along its edges and define the total weight of a

[2]However, note that the constant hidden in the asymptotic is smaller than for the other algorithms.

Algorithm 7: Forward substitution.

```
input  : Matrix $L \in S^{[n]\times[n]}$ with $L_{i,j} = 0$ for $i \leqslant j$, vector $b \in S^{[n]}$.
output: Solution of the system $x = L \cdot x + b$ (written to $b$).

for $i = 2 \dots n$ do
    for $j = 1 \dots (i-1)$, do
        $b_i := b_i + L_{i,j} \cdot b_j$
    end
end
return $b$
```

Algorithm 8: Backward substitution.

```
input  : Matrix $U \in S^{[n]\times[n]}$ with $U_{i,j} = 0$ for $j \leqslant i$, vector $b \in S^{[n]}$.
output: Solution of the system $x = U \cdot x + b$ (written to $b$).

for $i = (n-1) \dots 1$ do
    for $j = (i+1) \dots n$ do
        $b_i := b_i + U_{i,j} \cdot b_j$
    end
end
return $b$
```

set of paths as their sum then $M^*_{i,j}$ describes the weight of the set of all finite paths from i to j in this graph. Dividing the matrix in blocks like above is equivalent to splitting the graph into the single vertex 1 (with self-loop labeled a) and a rest graph with adjacency matrix D. Every path in this reduced graph corresponds to a word over the alphabet $\{a, b, c, D\}$. Consider for instance an arbitrary path p from 1 to itself. Either p only involves the self-loop $1 \to 1$, i.e. either it is a word in a^*, or it involves some b and c transitions [3]. In the latter case consider the first occurrence of b and the last occurrence of c in p. Thus, $p \in a^*bwca^*$ with some $w \in \Sigma^*$ describing a path from 2 to itself. By partitioning w into the paths that visit 2 exactly once, we see that $w \in (D + ca^*b)^*$. Continuing this reasoning we get the matrix M^*, where $M^*_{i,j}$ describes all paths from i to j. If we let $X^* := (D + ca^*b)^*$ describe all paths from the macro-vertex 2 to itself we obtain (cf. Section 4.1.3)

$$\begin{aligned}
M^* &= \begin{bmatrix} a^* + a^*bX^* & a^*bX^* \\ X^*ca^* & X^* \end{bmatrix} \\
&= \begin{bmatrix} 1 & a^*b \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} a^* & 0 \\ 0 & X^* \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ ca^* & 1 \end{bmatrix} \\
&= \underbrace{\begin{bmatrix} 0 & a^*b \\ 0 & 0 \end{bmatrix}^*}_{=:U^*} \cdot \underbrace{\begin{bmatrix} a^* & 0 \\ 0 & X^* \end{bmatrix}}_{=:D^*} \cdot \underbrace{\begin{bmatrix} 0 & 0 \\ ca^* & 0 \end{bmatrix}^*}_{=:L^*} .
\end{aligned}$$

Now we can continue the decomposition procedure inductively for the matrix X. Altogether, we obtain strictly upper triangular matrices $U_1, \dots, U_{n-1}$, strictly lower triangular matrices $L_1, \dots, L_{n-1}$, and diagonal matrices $D_1, \dots, D_{n-1}$ with

$$\begin{aligned}
M^* &= \prod_{i=1}^{n-1} U_i^* \cdot \prod_{i=1}^{n-1} D_i^* \cdot \prod_{i=1}^{n-1} L_i^* \\
&= \left(\sum_{i=1}^{n-1} U_i\right)^* \cdot \left(\sum_{i=1}^{n-1} D_i\right)^* \cdot \left(\sum_{i=1}^{n-1} L_i\right)^*
\end{aligned}$$

The last equality holds since U_i and L_i are nilpotent for all i, i.e. $U_i^2 = L_i^2 = 0$ and hence we have (e.g. for U_i)

$$\left(\sum_{i=1}^{n-1} U_i\right)^* = \prod_{i=1}^{n-1}(1 + U_i) = \prod_{i=1}^{n-1} U_i^*.$$

Finally, it is easy to see that $\prod_{i=1}^{n-1} D_i^* = \left(\sum_{i=1}^{n-1} D_i\right)^*$.

Note that the non-zero entries of all matrices are disjoint and hence they can all be stored in M itself, making the algorithm in-place (as opposed to the divide-and-conquer algorithm).

[3]Note that we abuse notation for the sake of readability and write a^* for the set $\mathcal{L}(a^*) = \{a\}^*$.

Algorithm 9: LDU-decomposition (slightly altered "Version 1" from [LRSS13]).

```
input  : Matrix $M \in S^{[n]\times[n]}$.
output: Matrices $L, D^*, U$ such that $M^* = U^*D^*L^*$ (stored in $M$ again).

for $i = 1 \ldots (n-1)$ do
    $M_{i,i} := M^*_{i,i}$
    /* Compute $b := a^*b$                                              */
    $M_{i,(i+1):n} := M_{i,i} \cdot M_{i,(i+1):n}$
    /* Compute $X = D + ca^*b$                                          */
    $M_{(i+1):n,(i+1):n} := M_{(i+1):n,(i+1):n} + M_{(i+1):n,i} \cdot M_{i,(i+1):n}$
    /* Compute $c := ca^*$                                              */
    $M_{(i+1):n,i} := M_{(i+1):n,i} \cdot M_{i,i}$
end
$M_{n,n} := M^*_{n,n}$ return $M$
```

Additions	$\frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6}$
Multiplications	$\frac{n^3}{3} + \frac{n^2}{2} - \frac{5n}{6}$
Kleene stars	n

Table 4.1: Number of operations needed for the LDU-decomposition of an $n \times n$ matrix via Algorithm 9.

Next, we count the number of additions, multiplications, and Kleene stars used by Algorithm 9. From the description, we see that it uses $(n-1)$ Kleene stars. For the number of additions, we observe that the matrix D is of size $(n-i) \times (n-i)$, and hence there are $(n-i)^2$ additions in each loop iteration. Altogether, the number of additions is given by

$$\sum_{i=1}^{n-1} (n-i)^2 = \sum_{i=1}^{n-1} i^2 = \frac{(n-1)n(2n-1)}{6} = \frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6}.$$

To count the multiplications we note that computing a^*b and ca^* each requires $(n-i)$ multiplications and the outer product ca^*b requires $(n-i)^2$ multiplications (since the term a^*b is already computed). Thus, the number of multiplications needed is given by

$$\sum_{i=1}^{n-1} (n-i)^2 + 2(n-i) = \sum_{i=1}^{n-1} i^2 + 2i = \frac{n^3}{3} + \frac{n^2}{2} - \frac{5n}{6}.$$

As observed in [LRSS13], we can reorder the steps in Algorithm 9 to make it similar to Algorithm 4.1.1 in [GVL96]. The idea is to avoid computing the outer product and sum

Additions	$\frac{n^3}{3} - n^2 + \frac{2n}{3}$
Multiplications	$\frac{n^3}{3} - \frac{n}{3}$
Kleene stars	n

Table 4.2: Number of operations needed for the LDU-decomposition of an $n \times n$ matrix via Algorithm 10.

$D + ca^*b$ in each loop iteration which touches every entry in the lower right submatrix and leads to bad caching behavior in practice. Instead we compute the matrix row by row so that each entry is computed in a "lazy" fashion. This row-wise approach detailed in Algorithm 10 is a slightly optimized variant of Algorithm 8 in [LRSS13] which operates column-wise (our version avoids the re-computation of the Kleene star $M^*_{i,i}$). We consider this row-wise version here, since matrices in FPSOLVE are also stored row-wise and hence this approach benefits from better caching performance in practice.

Contrary to the statement in [LRSS13], Algorithm 10 actually needs *less* operations than Algorithm 9. In particular it needs n Kleene stars, a number of multiplications given by

$$\sum_{i=1}^{n} \frac{(i-1)(i-2)}{2} + (i-1) + (i-1)(n-i) + (n-i) = \frac{n^3}{3} - \frac{n}{3}$$

and

$$\sum_{i=1}^{n} \frac{(i-1)(i-2)}{2} + (i-1)(n-i) = \frac{n^3}{3} - n^2 + \frac{2n}{3}$$

many additions. To obtain the total number of operations required to solve the thee linear systems

$$z = Lz + b \quad y = Dy + z \quad x = Ux + y$$

note that we can solve the diagonal system $y = Dy + z$ with n multiplications (as the matrix D is already starred). Additionally, each of the (forward/backward) substitutions uses $\frac{n(n-1)}{2}$ additions and multiplications.

We conclude this section with a detailed summary of the complexity for solving linear systems in Table 4.3. To assess the complexity of the methods that compute $A^* \cdot b$ directly we have to account for the complexity of computing A^* and an additional matrix-vector multiplication.

It is quite remarkable that Floyd-Warshall and the recursive divide-and-conquer algorithm both need roughly three times more additions and multiplications than the LDU-decomposition.

Algorithm 10: LDU-decomposition (an optimized row-wise variant of "Version 2" from [LRSS13]).

```
input  : Matrix $M \in S^{[n]\times[n]}$.
output: Matrices $L, D^*, U$ such that $M^* = U^*D^*L^*$ (stored in $M$ again).

for $i = 1 \dots n$ do
    /* $v$ is used to build the $i$-th row of $L$.                    */
    $v := M_{i,1:i}$
    for $k = 1 \dots (i-1)$ do
        for $l = (k+1) \dots (i-1)$ do
            $v_l := v_l + v_k \cdot M_{k,l}$
        end
    end
    for $j = 1 \dots (i-1)$ do
        /* The diagonal entries are already starred.                    */
        $M_{i,j} := v_j \cdot M_{j,j}$
    end
    $M_{i,i} := v_i^*$
    for $k = 1 \dots (i-1)$ do
        for $l = (i+1) \dots n$ do
            $M_{i,l} := M_{i,l} + v_k \cdot M_{k,l}$
        end
    end
    for $l = (i+1) \dots n$ do
        $M_{i,l} := M_{i,i} \cdot M_{i,l}$
    end
end
return $M$
```

Algorithm	# Additions	# Multiplications	# Kleene stars
Floyd-Warshall	$n^3 - n^2$	$n^3 + n^2 - n$	n
D & C	$n^3 - n^2$	$n^3 + n^2 - n$	n
LDU (Version 1)	$\frac{n^3}{3} + \frac{n^2}{2} - \frac{n}{3}$	$\frac{n^3}{3} + \frac{3n^2}{2} - \frac{5n}{6}$	n
LDU (Version 2)	$\frac{n^3}{3} - \frac{n}{3}$	$\frac{n^3}{3} + n^2 - \frac{n}{3}$	n

Table 4.3: Number of operations required by different approaches to solve a system of right-linear equations $x = Ax + b$ over an ω-continuous semiring.

4.1.4 Symbolic Solving

Consider again the following algebraic system over a commutative semiring

$$X = XX + c.$$

As we have seen, Newton's method applied to this equation yields

$$X^{<d} = X^{<d-1} + X^{=d-1} \qquad X^{=d} = 2 \cdot X^{<d-1}X^{=d} + X^{=d-1}X^{=d-1}$$

So every iteration of Newton's method essentially consists of solving the linear equation

$$X^{=d} = 2 \cdot X^{<d-1}X^{=d} + X^{=d-1}X^{=d-1}$$

for different values of $X^{<d-1}$ and $X^{=d-1}$ that have been computed in previous steps.

The symbolic solution of this equation (viewing $X^{<d-1}$ and $X^{=d-1}$ as symbolic constants) is given by

$$X^{=d} = \left(2 \cdot X^{<d-1}\right)^* \cdot \left(X^{=d-1}\right)^2.$$

Hence, *solving* the system in each iteration reduces to *evaluating* a symbolic expression. Technically, symbolic solving means to "un-interpret" the Jacobian of the algebraic system as terms (or formally, as elements of the free semiring) and then solving the system over this term-semiring.

To better illustrate the potential benefit of symbolic solving consider the generic two dimensional equation of the form $X = AX + b$:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}.$$

The solution of this system can be explicitly computed as $X = A^* \cdot b$ (e.g. via the divide-and-conquer method) which gives

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} (\mathbf{a}^*\mathbf{b}(c\mathbf{a}^*\mathbf{b} + d)^*ca^* + a^*)e + \mathbf{a}^*\mathbf{b}(c\mathbf{a}^*\mathbf{b} + d)^*f \\ (c\mathbf{a}^*\mathbf{b} + d)^*ca^*e + (c\mathbf{a}^*\mathbf{b} + d)^*f \end{pmatrix}.$$

Note that several subterms in this expression appear repeatedly, e.g. the highlighted term $\mathbf{a}^*\mathbf{b}$. Over some semirings (e.g. the semilinear sets – see Chapter 5) computing the semiring operations is costly and hence, recomputing the value of expressions leads to unnecessary complexity. Computing a symbolic solution enables us to detect such common subexpressions, store them in a compressed fashion, and hence avoid to re-evaluate expressions. For example the above solution to the two-dimensional equation can be represented as the "syntax-DAG" in Figure 4.1.

Although symbolic solving significantly reduces the number of semiring operations needed, the overhead from computing, storing, and evaluating the symbolic solution is not always negligible. This is particularly true for numeric semirings (like the semiring of nonnegative reals). For these semirings the operations can be computed so fast that the additional overhead of storing and evaluating a symbolic solution outweighs the benefits.

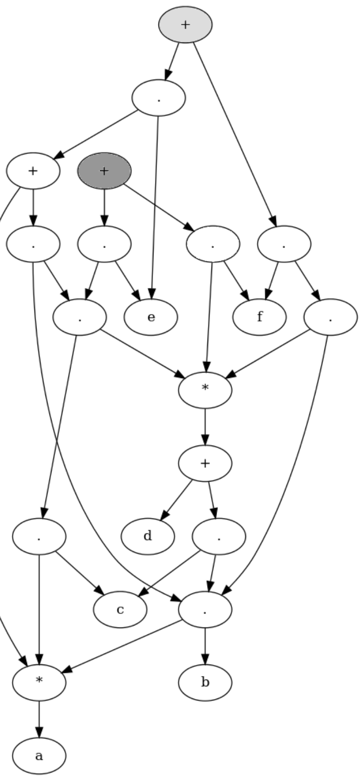


Figure 4.1: Succinctly representing all terms of the matrix-vector product $\binom{x}{y} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}^* \cdot (e, f)^T$ using BDD-like sharing of subexpressions. The x-component corresponds to the topmost node colored in light gray. The y-component is colored dark gray. By reversing the direction of all edges this can be read as an arithmetic circuit with output gates colored in gray

4.1.5 Decomposition into SCCs

Especially in large algebraic systems equations can often be solved independently of one another and hence we can decompose the system into independent parts. More precisely, we define a dependency relation on the set of variables $\mathcal{X} = \{X_1, \dots, X_n\}$ as follows: X_i *depends on* X_j if X_j occurs in the defining equation F_i. To determine the solution of a variable X it then suffices to determine the values of all variables on which X depends.

The depends-on-relation can be viewed as a (directed) dependency graph of the system (with nodes $\mathcal{X}$). We can decompose this graph into its strongly connected components (SCCs) and sort the SCCs in reverse topological order. Finally, we solve the corresponding equations in a bottom-up fashion. This decomposition was originally proposed by Etessami and Yannakakis for the analysis of recursive Markov chains [EY05].

4.2 FPSOLVE: A Generic Library for Algebraic Systems

In this section we present FPSOLVE, a generic library implementing the algorithms we have described before. First, we survey its features (Section 4.2.1) and give some examples how to use the standalone solver included in FPSOLVE (Section 4.2.2). We then describe further implementation details (Section 4.2.3) and report on the performance of our implementation (Section 4.2.4).

4.2.1 Overview and Architecture

FPSOLVE is written in C++ and comprises roughly $9,000$ lines of code. Our code makes heavy use of templates to enable both generic and efficient code via compile-time polymorphism. The drawback of this is that our library takes some time to compile. In our code we also try to use features of the new C++11 standard (like move-semantics, lambda-expressions) where appropriate to make the code more readable and efficient. The FPSOLVE library consists of three main parts:

- Data structures (polynomials, matrices, BDD-like DAG-structure for free terms)
- Semirings (semilinear sets, non-negative rationals, why semiring, generic product semiring, . . .)
- Solving algorithms (fixpoint iteration, various forms of Newton's method using different methods to solve linear equations)

Additionally, we have built two small applications demonstrating the use of the library:

- A solver for algebraic systems over a chosen semiring.
- A tool that tests equivalence of context-free grammars by interpreting them as systems of equations over various idempotent semirings and comparing their solutions.

FPSOLVE depends on several external libraries and frameworks. Some of these are not essential to build the core library but offer additional features.

- CPPUNIT for unit-tests: every class of the library should be accompanied by a test-class. We also wrote generic unit tests that should pass for any semiring. These come very handy when developing and integrating new semirings.
- BOOST for parsing (`Boost::spirit`) and other I/O-tasks (like program argument handling).
- GMP for arbitrary precision numbers (e.g. used to implement the semiring $\mathbb{Q}_{\geqslant 0} \cup \{\infty\}$).

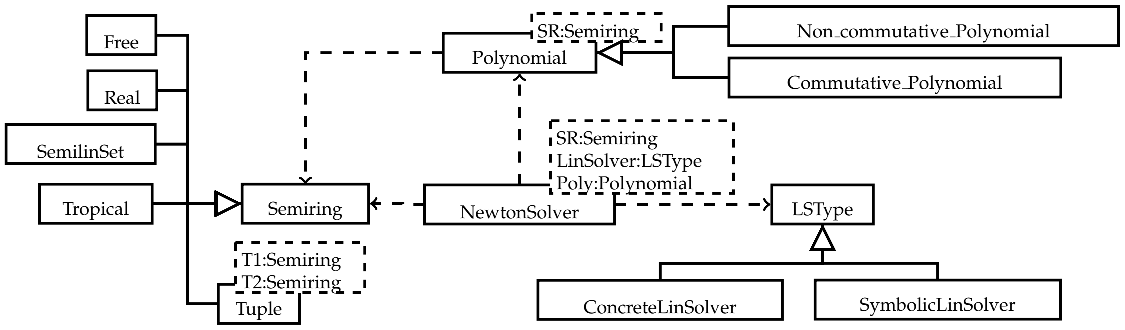


Figure 4.2: A (simplified) part of FPSOLVE's architecture.

- optional: GENEPI (using MONA or LASH) for representing semilinear sets by NDDs.
- optional: LIBFA for representing lossy semirings (or equivalently subword-closed languages) by finite automata (see Chapter 7).

FPSOLVE is free software (distributed under the BSD license) and can be obtained from `https://github.com/mschlund/FPsolve`.

In Figure 4.2 we show the most important part of the architecture of FPSOLVE as a simplified UML class diagram[4]. Many classes are templated, e.g. `Polynomial` depends on a `Semiring` template parameter. The dashed arrows between the classes indicate the dependency relation: An arrow from A to B means that the behavior of A might change if B is altered. At the heart of the diagram is the `NewtonSolver` class which provides a generic function `solve_fixpoint` that performs a desired number of Newton iterations on a system of polynomials and returns a vector of approximations.

4.2.2 Using the Standalone Solver

FPSOLVE is mainly designed as a library to facilitate writing tools that make use of Newton's method over semirings. As an example of such a tool FPSOLVE features a standalone solver that can solve algebraic systems over various semirings

To apply the solver, one has to describe the algebraic system as a BNF-style context-free grammar. Variables of the system are enclosed in angle brackets, the addition $x + y$ is written as $x \mid y$, and multiplication is written implicitly by juxtaposition of terms.

Consider the following system from Chapter 3 over the reals

$$X = 0.2XY + 0.8 \qquad Y = 0.5XYY + 0.5.$$

To approximate the least solution using our tool we create a text file `test.g` containing:

[4]We have omitted several other template parameters that are not relevant for the discussion. E.g. the class of semilinear sets gets a simplifier class as parameter that is used to simplify the set after each operation.

```
<X> ::= 0.2 <X> <Y>  | 0.8;
<Y> ::= 0.5 <X> <Y> <Y> | 0.5 <X>;
```

The simplest invocation of the tool is

```
$ ./fpsolve -f test.g --float
```

This minimal set of parameters specifies

- the input file (`-f test.g`)
- the semiring over which to interpret the system. Here we use the (extended) reals $\mathbb{R}_{\infty}^{\geqslant 0}$ represented via machine precision floating point numbers (`--float`).

The solver produces the following output

```
Solver: Newton Concrete
Iterations: 3
Solving time: 0 ms (185us)
X == 0.949488
Y == 0.789919
```

The output reports

- The solving algorithm used: the default used is Newton's method with concrete solving of the linear system, i.e. we invert the evaluated Jacobian in every iteration.
- The number of iterations. The default for n equations is $n + 1$ iterations since over commutative and idempotent semirings this is sufficient to compute the solution [EKL07b].
- The time in milliseconds (and microseconds) needed to run the stated number of iterations.
- The value of the approximation computed.

We can also manually set the number of iterations (using the switch `-i`) and the solving algorithm (via the switch `-s`). Current choices for the solving algorithms are:

- Fixpoint iteration: `kleene`
- Newton's method using concrete or symbolic solving for the linear system:
  - `newtonConc` or `newtonSymb` computes the Kleene star of via the divide-and-conquer method (concrete or symbolic)
  - `newtonCLDU` or `newtonSLDU` solves the linear system using the LDU decomposition (concrete or symbolic)

- Newton's method for "numeric" semirings (`newtonNumeric`) which provide a subtraction operation (currently only `--float` and `--rat`). This method uses the LDU decomposition for solving linear equations and computes $\delta_d = F(\nu_d) - \nu_d$ as explained in Section 3.4.3.

For example we can call the tool like

```
$ ./fpsolve -f test.g --float -s newtonNumeric -i 6
```

which produces the output

```
Solver: Newton Numeric (Float)
Iterations: 6
Solving time: 0 ms (139us)
X == 0.96
Y == 0.833333
```

As we can see, with 6 iterations we already obtain a very good approximation to the true solution $(X, Y) = (0.96, 0.83\overline{3})$. Note that the X-component seems exact because of rounding errors.

To get the exact value of the Newton iterates (without rounding errors) we can solve the same system over the rational numbers which are implemented using the GMP library for arbitrary precision arithmetic. To this end we create a file `test2.g` containing

```
<X> ::= 1/5 <X> <Y>  | 4/5;
<Y> ::= 1/2 <X> <Y> <Y> | 1/2;
```

and we call our tool with

```
$ ./fpsolve -f test.g --rat -s newtonNumeric -i 3
```

which produces

```
Solver: Newton Numeric (Rat)
Iterations: 3
Solving time: 0 ms (207us)
X == 1391/1465
Y == 15044/19045
```

Note that the tool outputs fractions with very large denominators for 6 iterations so we do not show them here.

For decomposing the system into SCCs as explained in Section 4.1.5, the tool offers the option `--scc`.

4.2.3 Implementation Details

Here we briefly survey the main data structures used by FPsolve and describe how to extend FPsolve with new semirings.

Data Structures

Variables FPSOLVE does not distinguish between variables and constants. All variables are maintained by a “variable pool” (which is essentially a factory class) that assigns a unique id to every variable and keeps the string representation of the variable separate. This way we can also generate new fresh variables which is needed e.g. for “un-interpreting” a polynomial when computing its symbolic unfolding.

Matrices Matrices are stored row-wise as a dense vector in a single contiguous block in memory. Currently, FPSOLVE does not use sparse matrices but this could easily be changed without affecting the rest of the library and could lead to performance gains, especially for larger systems.

Our implementation provides different algorithms to compute the Kleene star of a matrix (Floyd-Warshall, recursive divide-and-conquer method), and to compute the LDU-decomposition for solving systems of linear equations (cf. Section 4.1.3).

Polynomials Polynomials come in two flavors in FPSOLVE, commutative and non-commutative ones. Both are derived from `Semiring` (but not from `StarableSemiring`, since the Kleene star is not defined on polynomials in general). As such they both offer addition and multiplication functions together with other ones like taking derivatives etc.

A commutative polynomial is implemented as a map from monomials to semiring elements and a (commutative) monomial is essentially a map from variables to their multiplicity (i.e. their exponent in the monomial).

Non-commutative polynomials are implemented as maps from (non-commutative) monomials to their multiplicity in $\mathbb{N}$. A non-commutative monomial is simply a term of the form $a_0X_1a_1\cdots a_nX_na_{n+1}$ where $X \in \mathcal{X}$ and $a_i \in S \setminus \{0\}$. Note that we cannot simply store such words in a list (since list elements have to have the same type). To get around this problem, we use *three* lists to store a monomial: The first and second list consecutively store the semiring values and variables occurring. Since we never store values $a_i = 1$ explicitly, we cannot uniquely reconstruct the term $a_0X_1a_1\cdots a_nX_na_{n+1}$ from these two lists alone (we just know that the term is a shuffle of the two sequences such that no two semiring values are adjacent). To reconstruct the word $a_0X_1a_1\cdots a_nX_na_{n+1}$, we use the third list that stores for each position whether the element is a variable or a semiring value together with a pointer to the index in the first or second lists, respectively.

Example 4.5. *Consider the monomials* $m_1 = aXYb$ *and* $m_2 = cYdYd$ *represented as three lists. The first list describes a word over* $\{S, V\}^*$ *indicating for each position of the monomial whether it is a semiring element or a variable. Note that the factor SS does not appear in this list since adjacent semiring elements are always reduced to one by multiplying them.*

a	b

c	d	d

X	Y

Y	Y

S	*V*	*V*	*S*

S	*V*	*S*	*V*	*S*

Their product is given by $m_1 \cdot m_2 = aXY(bc)YdYd$ *which is represented by the lists depicted below:*

a	bc	d	d

X	Y	Y	Y

S	*V*	*V*	*S*	*V*	*S*	*V*	*S*

This encoding of monomials seems a bit complicated at first, but has the advantage that multiplication of two monomials is reduced to simple list concatenation together with a reduction step to merge two semiring elements at the border. Altogether, this approach allows to multiply two non-commutative monomials using at most one multiplication of semiring elements.

Shared Structure for Terms Using the symbolic solving method described in Section 4.1.4 for solving linear systems we have to be able to compute with free terms (comprising uninterpreted constants, addition, multiplication, and Kleene star) and store them efficiently. Formally, the set of "free terms" constitutes the *free semiring* which is also implemented in FPSOLVE.

The *free semiring* over an alphabet Σ consists of all terms $\mathcal{T}_\Sigma$ (built from $+, \cdot, ^*$), inductively defined via

- Basic terms are 0, 1, and a for every $a \in \Sigma$.
- If t_1, t_2 are terms then $(t_1 + t_2)$, $(t_1 \cdot t_2)$, and $(t_1)^*$ are terms.

To turn this term algebra into a semiring, we define equivalence on terms $t_1 \equiv t_2$ if the identity $t_1 = t_2$ holds in all semirings (cf. Section 2.2). Hence the semiring can be defined as $\mathcal{T}_\Sigma / \equiv$, the set of terms modulo the equivalence $\equiv$.

Example 4.6. *For* $\Sigma = \{a, b, c\}$ *valid terms are* $a + a$, $0 + (1 + 0) + 1 \equiv 1 + 1$, *or*

- $(a + b) \cdot (c + 0) \equiv ac + bc$ *by distributivity and definition of the neutral element* 0.

- $(a+b)^* \equiv (a^*b)^*a^*$ *since this holds in any (inductive) semiring (cf. Section 2.2).*

Since any term in $\mathcal{T}_\Sigma$ can be viewed as a rational expression we can identify the free semiring over Σ with $\mathbb{N}_\infty^{\mathrm{Rat}}\langle\langle\Sigma^*\rangle\rangle$, the rational power series with coefficients in $\mathbb{N}_\infty$. Note that $\mathbb{N}_\infty^{\mathrm{Rat}}\langle\langle\Sigma^*\rangle\rangle$ is not ω-continuous but is embedded into the ω-continuous semiring $\mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle$. Hence, systems of polynomial equations over $\mathbb{N}_\infty^{\mathrm{Rat}}\langle\langle\Sigma^*\rangle\rangle$ do not necessarily have solutions in $\mathbb{N}_\infty^{\mathrm{Rat}}\langle\langle\Sigma^*\rangle\rangle$ but only in $\mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle$. However, the elements of $\mathbb{N}_\infty\langle\langle\Sigma^*\rangle\rangle$ are not finitely representable.

There are many possibilities to represent elements from $\mathbb{N}_\infty^{\mathrm{Rat}}\langle\langle\Sigma^*\rangle\rangle$, such as rational expressions or weighted finite automata [DKV09]. In FPSOLVE we choose the representation as rational expressions but store them concisely as a directed acyclic graph (DAG) similar to BDDs [Bry86, And98] as described before in Section 4.1.4. Technically, we keep a global map of (sub-)expressions and create new ones only when needed. We do not reduce expressions once they have been generated, but we apply the obvious reductions at generation time, e.g. $x + 0 = x$ or $1 \cdot x = x$.

Extending FPSOLVE

An important requirement in the design of FPSOLVE was to make additions to the library easy to implement (e.g. new semirings or new methods to solve linear equations). All algorithms work out of the box with new semirings.

All new semirings should be derived from the abstract class `StarableSemiring` and are required to provide (at least) the following functions:

- Operators += and $\star$= used to add and multiply semiring values.
- A function `star()` to compute the Kleene star of elements.
- A constructor taking a `string` argument. This constructor is used to parse string representations of semiring values (e.g. when using the standalone solver).

The operators + and $\star$ are implemented in the abstract class `StarableSemiring` using the (virtual) operators += and $\star$=.

4.2.4 Performance of FPSOLVE

Here, we briefly study the performance of FPSOLVE, its scalability and show that FPSOLVE implements Newton's method with a running time that matches the theoretical analysis.

To this end, it is important to actually measure the performance of the core system instead of the performance w.r.t. a particular semiring implementation. One possibility to eliminate any semiring-specific bias is to simply report the number of semiring operations executed. However, this measure would only reflect the complexity of the general algorithms and not quantify the overhead of the implementation at all.

Hence, we study the performance of FPSOLVE for systems over the nonnegative reals $\mathbb{R}^{\geqslant 0}$ that are implemented using machine-precision floating point numbers. This way, every semiring operation roughly takes constant time and thus, this experiment provides a way to assess the performance of FPSOLVE.

We randomly generated quadratic equations over $[0, 1]$ and record the running time needed to approximate the solution using variants of Newton's method. [5] As we are only interested in how the performance varies with the size of the system we fixed the number of Newton iterations to 10. Each equation has $\varepsilon\binom{n}{2}$ monomials and we vary the "density" ε from 0.1 to 0.5. Note that these systems are rather dense and large, e.g. the textual description of the system with 100 variables and density 0.5 needs 7.6 MB. We compiled FPSOLVE using gcc 4.8 with optimizations (`-O2`) and ran it on a machine with an Intel I7 CPU with 2.5 GHz and 16 GB of memory. During the experiments memory usage never exceeded a few megabytes.

We recorded the time needed to execute 10 Newton iterations with FPSOLVE's standalone solver (options `--float -i 10`) using three different algorithms. The results are shown in Figure 4.3: On the top, we show the performance of the default concrete solving method which inverts the evaluated Jacobian in every iteration (option `-s newtonConc`). In the middle row, we solve the concrete system via LDU decomposition (option `-s newtonCLDU`). On the bottom, we use LDU decomposition and additionally compute δ_d by subtraction (option `-s newtonNumeric`).

As expected from our theoretical analysis, LDU decomposition is (slightly) faster than computing the Kleene star of the matrix but surprisingly does not lead to large performance gains. However, the improved computation of δ_d over $\mathbb{R}^{\geqslant 0}$ via $\delta_d = F(\nu_d) - \nu_d$ leads to a significant speedup. This suggests that there is still some room for improving the computation of δ_d in the general case.

For quadratic systems we expect an asymptotically cubic running time (since the number of Newton steps is constant). Thus, if we normalize the runtime data by n^3 we expect the to see a constant trend (at least for larger n). The right column in Figure 4.3 seems to support this hypothesis since at about $n = 60$ the curves level off.

4.3 Conclusions

In this chapter we have described general algorithmic techniques used to implement Newton's method over semirings. Furthermore, we have presented FPSOLVE, a generic and efficient C++ implementation of these algorithms for solving fixpoint equations. FPSOLVE constitutes the first implementation of Newton's method generalized to ω-continuous semirings. Our implementation is parametric in the semiring and can be easily extended with new semirings and solving algorithms. Our work can be seen as an

[5] These benchmarks are available at `https://github.com/mschlund/newton/tree/master/c/test/grammars/float-random`.

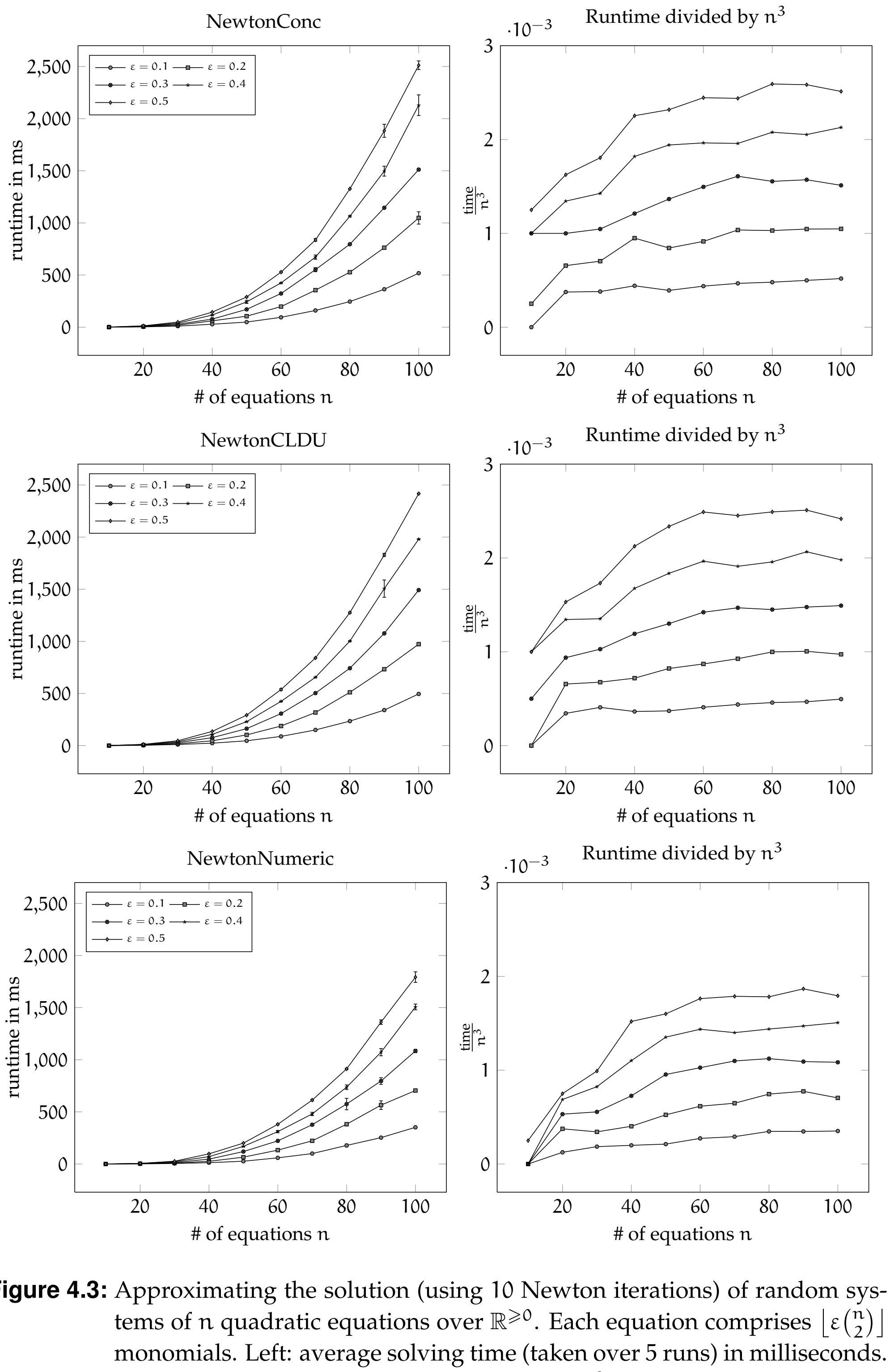


Figure 4.3: Approximating the solution (using 10 Newton iterations) of random systems of n quadratic equations over $\mathbb{R}^{\geqslant 0}$. Each equation comprises $\lfloor \varepsilon \binom{n}{2} \rfloor$ monomials. Left: average solving time (taken over 5 runs) in milliseconds. Right: numbers from the left normalized by n^3.

independent contribution to the research program outlined by Litvinov et al. [LRSS13] who advocate the use of generic algorithms for solving problems in tropical and idempotent mathematics and suggest to implement a generic library of such algorithms [6].

Related Work PREMO (Probabilistic Recursive Model analyzer) is a tool (written in Java) for analyzing recursive Markov chains, probabilistic context-free grammars, and polynomial equations over the nonnegative reals [WE07]. PREMO is only applicable to systems over the real numbers but offers several choices for solving linear equations (direct methods like Gaussian elimination or iterative methods like the CG-algorithm) and efficient data structures (e.g. sparse matrices) for handling large systems. Preliminary experimentation with PREMO suggests that its performance is worse than FPSOLVE for dense equations (maybe due to the use of Java) but better for sparse systems (due to data structures for sparse matrices). However, since PREMO is not open source and only offers a graphical user interface (which makes experiments cumbersome to perform) we cannot present a reliable comparison with FPSOLVE here.

The Weighted Pushdown Systems Library and Weighted Automata Library (cf. [RSJM05, SRJ, DTR]) offer algorithms for the analysis of semiring-weighted pushdown systems and weighted finite automata. However, algorithms for weighted pushdown systems are usually based on fixpoint iteration and are thus restricted to semirings that satisfy the ascending chain condition.

GOBLINT [ASV13, VV] is a static analyzer for multi-threaded C programs, specifically for detecting data races. For the analysis it uses variants of fixpoint iteration, thus the abstract domains either have to satisfy the ascending chain condition or widening must be employed to ensure termination.

Future Work We have several ideas for improvements to FPSOLVE that we want to implement in future work (besides adding more semirings and tweaking the generic algorithms):

- Large algebraic systems occurring in practical applications are often sparse. To enable efficient solving of such systems we want to incorporate data structures for sparse matrices into FPSOLVE which should be straightforward to incorporate.
- One idea to improve the efficiency in practice is to make FPSOLVE multi-threaded so we can take advantage of modern multicore processors. For example this would allow us to solve independent SCCs of an equation system efficiently using multiple threads.
- Computing the Kleene star of matrices is a problem well suited for parallelization [BGB10], but a generic parallel implementation (for arbitrary semirings) does not yet exist to the best of our knowledge.

[6] We thank Folkmar Bornemann for recently pointing us to their work.

5 Algorithms and Data Structures for Semilinear Sets

In this chapter we focus on algorithms and data structures for the semiring of semilinear sets. More specifically, we present two different representations for the elements of this semiring, discuss the properties of these representations and describe some optimization techniques we use in FPSOLVE.

After presenting some theoretical background on semilinear sets, we describe in Section 5.2 an explicit vector representation of semilinear sets implemented in FPSOLVE.

Then in Section 5.3 we present a symbolic representation of semilinear sets using number decision diagrams (NDDs) which we also implemented in FPSOLVE. In Section 5.4 we present a grammar testing tool implemented using the FPSOLVE library which makes use of these representations to check in-equivalence of context-free grammars modulo commutativity.

The results presented in this chapter have appeared in [STL13, ELS14b, ELS15]. The implementation of the symbolic NDD representation constitute the Master's thesis of Kerscher [Ker14].

5.1 Theoretical Background

Since Newton's method converges in $n + 1$ steps over any commutative idempotent semiring [EKL07b] it is highly desirable to study data structures for representing those semirings and algorithms to compute with them. The most general commutative idempotent semiring is $\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle$, i.e. for any commutative idempotent semiring S any ω-

continuous homomorphism $h : \Sigma \to S$ extends uniquely to an ω-continuous homomorphism $\tilde{h} : \mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle \to S$.

In the same way as $\mathbb{B}\langle\langle\Sigma^*\rangle\rangle$ is isomorphic to the (ω-continuous) language semiring $(2^{\Sigma^*}, \cup, \cdot, \emptyset, \{\varepsilon\})$ its commutative image $\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle$ is isomorphic to the (ω-continuous) semiring of Parikh-images of languages $(2^{\mathbb{N}^{|\Sigma|}}, +, \cdot, \emptyset, \{\vec{0}\})$ with the semiring operations on $X, Y \in 2^{\mathbb{N}^{|\Sigma|}}$ defined as

$$\begin{aligned} X + Y &:= X \cup Y \\ X \cdot Y &:= \{x + y : x \in X, y \in Y\} \end{aligned}$$

These definitions are inherited from the semiring of formal languages via the Parikh-image-homomorphism: For $A, B \subseteq \Sigma^*$ let $X := \mathsf{P}(A), Y := \mathsf{P}(B)$, then $\mathsf{P}(A \cdot B) = \{\mathsf{P}(u) + \mathsf{P}(v) : u \in A, v \in B\} = \mathsf{P}(A) \cdot \mathsf{P}(B)$.

Since elements in $\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle$ can be seen as the commutative images of arbitrary (e.g. also non-recursive) languages in Σ^*, we cannot hope to represent each element of $\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle$ finitely. However, since the Newton approximations of an algebraic system over $\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle$ are rational, it suffices for us to consider the sub-semiring $\mathbb{B}^{\text{rat}}\langle\langle\Sigma^{\oplus}\rangle\rangle$ of rational subsets. These can be represented finitely by rational expressions, i.e. terms built from atomic expressions $a \in \Sigma$ using addition, multiplication, and Kleene star. We call a term $p = \prod_{i \in [n]} a_i$ for $a_i \in \Sigma$ a *product*. An expression l is called *linear* if it is of the form $l = c \cdot \prod_{i \in [n]} p_i^*$ where p_i are products. A *semilinear* expression $s = \sum_{i \in [m]} l_i$ is a finite sum of linear expressions l_i.

As observed by [Pil73] we can use the following identities to transform each rational expression over an ω-continuous commutative idempotent semiring into a semilinear expression by "pushing Kleene stars down in the term structure".

Proposition 5.1 ([Pil73, Koz96]). *Let S be a commutative and idempotent ω-continuous semiring. Then the following identities hold:*

1. $(x^*)^n = x^*$ *for all* $n \in \mathbb{N} \setminus \{0\}$
2. $(x + y)^* = x^* \cdot y^*$
3. $(xy^*)^* = 1 + xx^*y^*$

Proof.

1. The case $n = 1$ is trivial and for $n = 2$ we get by means of the Cauchy product

$$(x^*) \cdot (x^*) = \left(\sum_{k \geqslant 0} x^k\right) \cdot \left(\sum_{k \geqslant 0} x^k\right) = \sum_{k \geqslant 0} \left(\sum_{i=0}^{k} 1\right) \cdot x^k \overset{\text{id.}}{=} \sum_{k \geqslant 0} x^k = x^*.$$

The case $n > 2$ then follows by induction.

2. For the second identity, we first use Lemma 2.3 and obtain

$$(x+y)^* = (x^*y)^*y^* = \sum_{n\geqslant 0}(x^*y)^n \cdot y^* \overset{\text{comm.}}{=} \sum_{n\geqslant 0}(x^*)^n y^n x^* \overset{\text{comm.+id.}}{=} x^* \cdot y^*.$$

3. The third identity follows by unrolling the Kleene star (cf. again Lemma 2.3):

$$(xy^*)^* = 1 + (xy^*)(xy^*)^* = 1 + \sum_{n\geqslant 0} xy^*(xy^*)^n = 1 + xy^* \sum_{n\geqslant 0} x^n = 1 + xx^*y^*.$$

□

Example 5.2.

$$\begin{aligned}(ab^*c + (ac^*)^*)^* &= (acb^* + 1 + aa^*c^*)^* \\ &= (acb^*)^* \cdot 1^* \cdot (aa^*c^*)^* \\ &= (1 + ac(ac)^*b^*) \cdot (1 + aa^*c^*) \\ &= 1 + ac(ac)^*b^* + aa^*c^* + a^2c(ac)^*a^*c^* \\ &= 1 + ac(ac)^*b^* + aa^*c^*\end{aligned}$$

where the last equality holds since $a^2c(ac)^*a^*c^*$ *is included in* aa^*c^*.

Remark 5.3 (Notation for Linear Expressions). *Let* $P = \{p_1, \dots, p_n\} \subseteq \Sigma^{\oplus}$ *be a finite set of (commutative) words. Then we set* $P^* = \prod_{i\in[n]} p_i^*$ *which is justified by interpreting* P *as the expression* $P = \sum_i p_i$. *This allows us to simplify the notation for linear expressions, so instead of* $L = c\prod_{i\in[n]} p_i^*$ *we write* $L = cP^*$.

In the following we write $[\![r]\!]$ to denote the set of vectors represented by a representation r (e.g. a rational expression). The linear (resp. semilinear) expressions over Σ (modulo commutativity) describe exactly the so called linear (resp. semilinear) subsets of vectors of $\mathbb{N}^{|\Sigma|}$:

Definition 5.4 (Linear and Semilinear Sets). *A subset* $L \subseteq \mathbb{N}^k$ *is called a* linear set *if* L *can be written as*

$$L = c + \sum_{i=1}^{n} \mathbb{N} \cdot p_i$$

where $c \in \mathbb{N}^k$ *and* $p_i \in \mathbb{N}^k$ *are called the* constant *and the* periods *of* L, *respectively.*

A subset $S \subseteq \mathbb{N}^k$ *is called* semilinear *if it can be written as a finite union of linear sets, i.e.*

$$S = \bigcup_{i=1}^{m} L_i$$

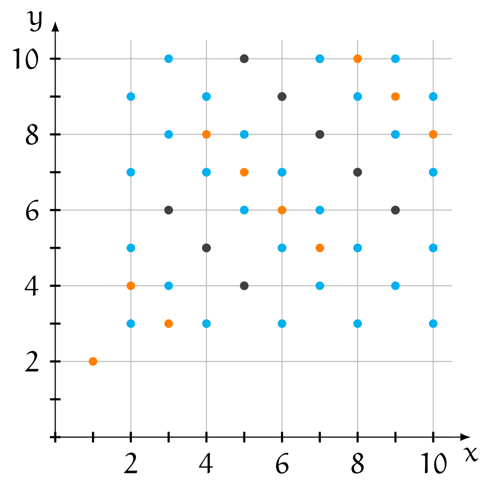
where each L_i *is a linear set.*

Note that the representation of semilinear sets by expressions is often convenient to prove identities (such as $(x+y)^* = x^*y^*$). On the other hand, viewing semilinear sets as subsets of $\mathbb{N}^k$ gives us access to the tools from linear algebra, geometry, and discrete optimization.

Example 5.5. *For* $k = 2$, *let*

$$L_1 = (1,2) + \mathbb{N}(1,2) + \mathbb{N}(2,1)$$
$$L_2 = (2,3) + \mathbb{N}(1,1) + \mathbb{N}(0,2) + \mathbb{N}(2,0)$$

and let $S = L_1 \cup L_2$ *denote a semilinear set. A part of the set* S *is shown in the following picture. Points* $p \in L_1 \setminus L_2$ *are colored orange,* $p \in L_2 \setminus L_1$ *are colored blue, and* $p \in L_1 \cap L_2$ *are colored dark-gray.*



If we let $\Sigma = \{a, b\}$ *and define the Parikh image (as the homomorphism induced) by* $\mathsf{P}(a) = (1,0)$ *and* $\mathsf{P}(b) = (0,1)$ *then the linear sets* L_1, L_2 *can be described as the Parikh images of regular languages (given by regular expressions):*

$$L_1 = \mathsf{P}(ab^2 \cdot (ab^2)^* \cdot (a^2b)^*)$$
$$L_2 = \mathsf{P}(ab \cdot (b^2)^* \cdot (a^2)^*)$$

Hence we obtain the representation of the semilinear set S *as (the Parikh-image of the language of) the expression:*

$$S = \mathsf{P}(ab^2 \cdot (ab^2)^* \cdot (a^2b)^* + ab \cdot (b^2)^* \cdot (a^2)^*).$$

5.2 Explicit Vector Representation

A straight-forward data structure to represent a linear set L is the pair (c, P) comprising the vector $c \in \mathbb{N}^k$ (the *constant*) and the set of *periods* $P = \{p_1, \ldots, p_n\}$ such that $L = c +$

$\sum_{i=1}^{n} \mathbb{N}p_i$. This is equivalent to representing L by the expression cP^*, but the geometric interpretation gives us more insight here.

At first sight, this representation is by no means unique, e.g. the set $L = (1,2) + \mathbb{N}(1,2) + \mathbb{N}(2,1)$ can be represented as the pair

$$((1,2), \{(1,2), (2,1)\})$$

or as

$$((1,2), \{(1,2), (2,1), (4,5)\}).$$

However, note that the vector $(4,5)$ in the second set is redundant, since it can be combined from the other two vectors as $(4,5) = 2 \cdot (1,2) + (2,1)$. We now show that redundancies like these are the only obstruction to uniqueness.

To this end, recall the partial order $\leqslant$ on $\mathbb{N}^k$ defined as

$$x \leqslant y :\Leftrightarrow \forall i \in [k] x_i \leqslant y_i.$$

Dickson's lemma [Dic13] states that $\leqslant$ is a well-founded partial order on $\mathbb{N}^k$, i.e. every set of vectors in $\mathbb{N}^k$ has a finite set of $\leqslant$-minimal elements. Linear sets are particularly simple in this regard:

Lemma 5.6. *A linear set $L \subseteq N^k$ has a unique $\leqslant$-minimal element.*

Proof. Let $L = c + \sum_{i=1}^{n} \mathbb{N}p_i$. Then every $x \in L$ can be represented as

$$x = c + \sum_{i=1}^{n} \lambda_i p_i$$

with $\lambda_i \in \mathbb{N}$. Hence $c \leqslant x$ for all $x \in L$. □

This means that it is possible to talk about *the* constant c of a linear set L.

Definition 5.7 (Cone, Basis). *Given a finite set $P = \{p_1, \ldots, p_n\} \subseteq \mathbb{N}^k$, we define its* cone *as*

$$\operatorname{cone}(P) := \left\{ \sum_{i=1}^{n} \lambda_i p_i : \lambda_i \in \mathbb{N} \right\}$$

A finite set $P = \{p_1, \ldots, p_n\}$ is called a basis *(of* $\operatorname{cone}(P)$*) if there is no $p \in P$ such that $p \in \operatorname{cone}(P \setminus \{p\})$.*

Thus, P is a basis if the vectors in P are "linearly independent" (if only coefficients from $\mathbb{N}$ are allowed), in particular we have $0 \notin P$. It is now easy to show that the pair (c, P) is a unique representation of the linear set $c + \operatorname{cone}(P)$ if P is a basis.

Proposition 5.8. *Let* $P = \{p_1, \dots, p_n\}$ *and* $Q = \{q_1, \dots, q_m\}$ *be bases (of* $\operatorname{cone}(P)$ *and* $\operatorname{cone}(Q)$*, respectively) and let* $c, c' \in \mathbb{N}^k$ *such that* (c, P) *and* (c', Q) *represent the same linear set* L*, i.e.*

$$L = c + \operatorname{cone}(P) = c' + \operatorname{cone}(Q).$$

Then $c = c'$ *and* $P = Q$.

Proof. The assertion $c = c'$ follows from the previous lemma.

Assume $P \neq Q$, so w.l.o.g. there is some $p_i \in P$ but $p_i \notin Q$. For every $q_j \in Q$ we have $c + q_j \in L$, so $q_j \in \operatorname{cone}(P)$ which means there is an index set $I_j \subseteq [n]$ such that

$$q_j = \sum_{k \in I_j} \lambda_{j,k} p_k$$

with *positive* $\lambda_{j,k} \in \mathbb{N} \setminus \{0\}$.

Now again since $c + p_i \in L$ we have $p_i \in \operatorname{cone}(Q)$, i.e. there is an index set $J_i \subseteq [m]$ and positive coefficients $\mu_j \in \mathbb{N} \setminus \{0\}$ with

$$p_i = \sum_{j \in J_i}^{m} \mu_j q_j = \sum_{j \in J_i}^{m} \mu_j \sum_{k \in I_j}^{n} \lambda_{j,k} p_k.$$

Since all coefficients in this sum are positive we must have $i \notin \bigcup_{j \in J_i} I_j$ and hence $p \in \operatorname{cone}(P \setminus \{p_i\})$ which contradicts the assumption that P is a basis. □

Remark 5.9 (Complexity). *The equivalence problem for linear sets is* NP*-complete since given two finite sets* P, P' *to decide whether* $\operatorname{cone}(P) = \operatorname{cone}(P')$ *is* NP*-complete: If* $\operatorname{cone}(P) = \operatorname{cone}(P')$ *then every* $p \in P$ *can be combined (with coefficients in* $\mathbb{N}$*) from vectors in* P' *and conversely. Each membership* $p \in \operatorname{cone}(P')$ *is witnessed by a vector* $\lambda_p \in \mathbb{N}^{|P'|}$ *and the size of* λ_p *is bounded by the size of* p*. All* λ_p*'s taken together provide a polynomial sized witness (w.r.t the size of* P *and* P'*) for* $\operatorname{cone}(P) = \operatorname{cone}(P')$.

On the other hand if we could decide $\operatorname{cone}(P) = \operatorname{cone}(P')$ *efficiently, we could solve the* NP*-complete* SUBSETSUM *problem efficiently. An instance of* SUBSETSUM *is given by a set* $A = \{a_1, \dots, a_n\} \subseteq \mathbb{N}$ *and a number* $b \in \mathbb{N}$*. By calling our procedure for deciding* $\operatorname{cone}(P) = \operatorname{cone}(P')$ *at most* $\mathcal{O}(n^2)$ *times starting with* $P = A$ *and* $P' = A \setminus \{a_i\}$ *we first reduce* A *to a basis* A' *for* $\operatorname{cone}(A)$*. Then we check if* $\operatorname{cone}(A') = \operatorname{cone}(A' \cup \{b\})$ *which is the case if and only if* $b \in \operatorname{cone}(A')$.

As a result, there is no polynomial algorithm to compute the unique minimal representation for a linear set (unless P = NP*).*

The explicit vector representation of semilinear sets S is as straightforward as for linear sets: S can be represented as a finite set of representations of linear sets $\{(c_1, P_1), \dots, (c_m, P_m)\}$. However, unlike in the case of linear sets this does not give us an easy canonical representation.

Example 5.10. *Consider the semilinear set* $S = L_1 \cup L_2 \cup L_3$

$$\begin{aligned} L_1 &= (1,2) + \mathbb{N}(1,1) + \mathbb{N}(0,1) \\ L_2 &= (2,1) + \mathbb{N}(1,1) + \mathbb{N}(1,0) \\ L_3 &= (2,2) + \mathbb{N}(1,1). \end{aligned}$$

Each linear set is uniquely represented by its constants and periods. Moreover the linear sets are disjoint. However, a "simpler" representation of S *can be given by* $S = L_1' \cup L_2'$

$$\begin{aligned} L_1' &= (1,2) + \mathbb{N}(0,1) \\ L_2' &= (2,1) + \mathbb{N}(1,0) + \mathbb{N}(0,1). \end{aligned}$$

Remark 5.11 (Complexity). *For semilinear sets (in our explicit vector representation), the equivalence problem is complete for* Π_p^2 *(the second level of the polynomial hierarchy) as shown by Huynh in [Huy80]. The proof of membership in* Π_p^2 *uses bounds for solutions of linear diophantine equations (similar to the proof of the fact that integer linear programming is in* NP*).*

5.2.1 Defining the Semiring Operations

We now define the semiring of semilinear sets in our explicit vector representation $(\mathcal{S}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$. To derive the definitions it is useful to consider the representations of semilinear sets as expressions first.

Let $S_1 = \sum_{i\in[n]} L_i$, $S_2 = \sum_{j\in[m]} M_j$ with linear expressions L_i, M_j. Then

$$\begin{aligned} S_1 + S_2 &= \sum_{i\in[n]} L_i + \sum_{j\in[m]} M_j \\ S_1 \cdot S_2 &= \sum_{(i,j)\in[n]\times[m]} L_i \cdot L_j \end{aligned}$$

The product of two linear expressions $cp_1^* \cdots p_n^*$ and $c'p_1'^* \cdots p_m'^*$ is given by the linear expression $(cc')p_1^* \cdots p_n^* p_1'^* \cdots p_m'^*$ (using commutativity to move the constants together).

The Kleene star of the expression S_1 can be computed using the identities $(x+y)^* = x^*y^*$ and $(xy^*)^* = 1 + xx^*y^*$ from Proposition 5.1.

$$S_1^* = \prod_{i\in[n]} L_i^* \qquad \text{with} \quad \left(c \prod_{i\in[l]} p_i^*\right)^* = 1 + cc^* \prod_{i\in[l]} p_i^* \text{ for linear expressions}$$

These simple observations can be translated to our vector representation as follows: Let $S_1 = \{L_1, \ldots, L_n\}$ and $S_2 = \{M_1, \ldots, M_m\}$ be two semilinear set representations, then we define

- $S_1 \oplus S_2 := S_1 \cup S_2$

- $S_1 \otimes S_2 := \{L_i \cdot M_j : i \in [n], j \in [m]\}$, where the product between two linear sets is defined as $(c, P) \cdot (c', P') := (c + c', P \cup P')$ (with $+$ the usual vector addition on $\mathbb{N}^k$).
- $\mathbf{0} := \emptyset$
- $\mathbf{1} := \{(0, \emptyset)\}$
- The Kleene star of a linear set (c, P) is the semilinear set $(c, P)^* := \mathbf{1} + \{(c, P \cup \{c\})\}$.
- Finally, the Kleene star of a semilinear set is defined as $S_1^* := \prod_{i \in [n]} L_i^*$.

Complexity Since the product of two linear sets $(c, P), (c', P')$ is given by $(c+c', P \cup P')$ it can be computed in time $\mathcal{O}(|P| + |P'|)$ (assuming constant-length vectors). The sum of two semilinear sets requires a single set union operation and thus takes linear time as well. Multiplication of semilinear sets requires us to perform all pairwise multiplication of linear sets and thus takes quadratic time

The computation of the Kleene star of a semilinear set S is more complex: if S contains n linear sets, computing S^* requires n multiplications of semilinear sets of the form $(1 + L)$. More precisely, let $S = L_1 + \dots + L_n$ with linear sets $L_i = (c_i, P_i)$. Then $S^* = \prod_{i=1}^{n} (1 + L_i')$ with $L_i' = (c_i, P_i \cup \{c_i\})$ which we can compute as

$$\begin{aligned} S^* &= 1 + \sum_{i \in [n]} L_i' + \sum_{\{i,j\} \in \binom{[n]}{2}} L_i' \cdot L_j' + \ldots + L_1' \cdots L_n' \\ &= \sum_{k=0}^{n} \sum_{I \in \binom{[n]}{k}} \prod_{i \in I} L_i' \end{aligned}$$

So in the worst-case computing S^* requires $\sum_{k=0}^{n} \binom{n}{k} - 1 = 2^n - 1$ additions and

$$\sum_{k=2}^{n} \binom{n}{k} (k-1) = (n-2) \cdot 2^{n-1} + 1$$

multiplications of linear sets (for $n \geqslant 2$)[1]. Hence, the computation of the Kleene star takes exponential time in the size of the input.

5.2.2 Optimizations

Some drawbacks of our explicit vector representation are

- The representation is very space consuming if implemented in a straightforward way.

[1]Recall that $\sum_{i=1}^{n} i\binom{n}{i} = n2^{n-1}$.

- The complexity of the multiplication and Kleene star operations is high (see above).

During experimentation with Newton's method instantiated for semilinear sets in FPSOLVE we noted that the semilinear sets occurring as Newton iterates carry a lot of redundancy, e.g. linear sets are not in their minimal form (i.e. there are period vectors that can be linearly combined from others). Furthermore, linear sets are often redundant since they are subsumed by others.

To address these issues, we reduce linear sets to their (unique) minimal representation (see above). Although this minimization of a linear set requires us to solve an NP-complete subset-sum problem (for vectors of dimension k) it is tractable in practice via dynamic programming since the entries in the vectors are typically small.

For semilinear sets our minimization procedure checks if any linear set is included in any other and hence is redundant. Note that we do not detect redundancies where some linear set is covered by two (or more) sets as in $L_1 \subseteq L_2 \cup L_3$ but $L_1 \nsubseteq L_2$ and $L_1 \nsubseteq L_3$.

To reduce the space consumption of our representation, we use a sparse vector representation and store each vector only once in memory. This is realized by a "vector pool" object. If we obtain a vector as a result of an operation this "pool" either allocates a new vector or returns the pointer to an already existing one. We also use the same technique to keep at most one copy of each linear set in memory.

Note that with these optimizations the space requirements of the semilinear sets are reduced significantly. Nonetheless, the concise representation does not help to reduce the time complexity of the operations. For instance, if we multiply two semilinear sets we still have to multiply all pairs of linear sets. As a result the memory requirements of our representation is almost negligible compared to the time complexity of the operations.

5.2.3 Over-Approximations

In several cases it can be sufficient to compute a sound over-approximation of a semilinear set. Consider for instance the problem to decide whether the set of runs of a system (given as a CFG) intersects a set of "bad" runs (e.g. specified by a CFG as well). The problem of testing the non-emptiness of an intersection of CFLs is a classical undecidable problem.

We can approach the problem by computing the commutative images of the runs of the system (as a semilinear set R) and the commutative image of the bad runs (as a semilinear set B). Then we check whether $B \cap R = \emptyset$. If the intersection is empty then our system is guaranteed to be safe. Otherwise we may have discovered a *spurious* bad run, i.e. a run of the system that is equal to a bad run only modulo commutativity. In this case we could refine our approximation, e.g. by excluding the spurious run from B and R.

However, the representation of R could be prohibitively large and hence we cannot even compute it in reasonable time. As a remedy we can compute a sound over-approximation, i.e. $\widetilde{R} \supseteq R$ which has a smaller representation and might be sufficient to show safety, i.e. $\widetilde{R} \cap B = \emptyset$. We have implemented two different over-approximations for semilinear sets.

GCD-Approximation For every linear set (c, P) we replace every vector $v \in P$ by $\frac{1}{d} \cdot v$ where $d = \gcd(v_1, \ldots, v_k)$ is the greatest common divisor of all entries in v. It is easy to see that this approximation preserves the direction of the period vectors: For a period $v \in P$ the set $\mathbb{N}v \subseteq \mathbb{N}^k$ describes a one dimensional discrete "line with gaps". The approximation fills these gaps with more integer points but does not change its slope, i.e. $\mathbb{N}v \subseteq \mathbb{N}v' \subseteq \mathbb{Q}v \cap \mathbb{N}^k$.

A similar but more imprecise approximation would compute the convex hull of each linear set and intersect it with $\mathbb{N}^k$.

Example 5.12. *Consider the semilinear set* $S = L_1 \cup L_2$ *represented as*

$$\begin{aligned} L_1 &= (2,1) + \mathbb{N}(4,2) + \mathbb{N}(1,0) \\ L_2 &= (4,2) + \mathbb{N}(2,1) \end{aligned}$$

The GCD-Approximation replaces the period $(4,2)$ *in* L_1 *by* $(2,1)$ *to obtain the over-approximation* $L_1' \supseteq L_1$.

$$L_1' = (2,1) + \mathbb{N}(2,1) + \mathbb{N}(1,0)$$

This in turn allows us to simplify the semilinear set further, since now $L_2 \subseteq L_1'$ *so* L_2 *can be omitted from* S.

Remark 5.13. *The result of the GCD-Approximation depends on the (syntactic) representation of a semilinear set.*

As a simple example consider the following two representations S_1, S_2 *of the same set of vectors* $[\![S_1]\!] = [\![S_2]\!]$:

$$\begin{aligned} S_1 &= \{(0,0) + \mathbb{N}(1,0)\} \cup \{(2,2) + \mathbb{N}(2,2) + \mathbb{N}(1,0)\} \\ S_2 &= \{(0,0) + \mathbb{N}(2,2) + \mathbb{N}(1,0)\} \end{aligned}$$

The corresponding GCD-approximations are

$$\begin{aligned} S_1' &= \{(0,0) + \mathbb{N}(1,0)\} \cup \{(2,2) + \mathbb{N}(1,1) + \mathbb{N}(1,0)\} \\ S_2' &= \{(0,0) + \mathbb{N}(1,1) + \mathbb{N}(1,0)\} \end{aligned}$$

which represent different sets of vectors since $(1,1) \notin S_1'$.

Multi-Linear Sets For a given semilinear set $S = \{(c_1, P_1), \dots, (c_n, P_n)\}$ we define its *multi-linear representation* as the pair $M = (\{c_1, \dots, c_n\}, \bigcup_{i \in [n]} P_i)$. A multi-linear set $M = (C, P)$ represents the set of vectors

$$[\![M]\!] := \{c + \sum_{p \in P} \lambda_p p : c \in C, \lambda_p \in \mathbb{N}\}$$

The intuition behind it is that we can choose any of the constants and then use any periods as in the case of linear sets. This approximation is precise if the period sets attached at different constants are the same. Otherwise this approximation still keeps "asymptotic upper/lower" bounds on the relationship of different components. Consider a semilinear set consisting of two linear sets with a single period vector: $(c_1, \{p_1\})$ and $(c_2, \{p_2\})$. The corresponding multi-linear abstraction would be $(\{c_1, c_2\}, \{p_1, p_2\})$. Clearly (unless $c_1 = c_2$) we add some "spurious" points by additionally admitting the line $c_1 + \mathbb{N}p_2$.

Like semilinear sets, multi-linear sets form a commutative semiring and the operations can be defined directly on their representation (C, P). As an invariant we require that the set C is always nonempty and add a special element $\perp$ to represent the empty set of vectors.

Let (C_1, P_1) and (C_2, P_2) be multi-linear sets, then we define

- $(C_1, P_1) \oplus (C_2, P_2) := (C_1 \cup C_2, P_1 \cup P_2)$ (and $(C, P) \oplus \perp = (C, P)$)
- $(C_1, P_1) \otimes (C_2, P_2) := (C_1 \cdot C_2, P_1 \cup P_2)$ (and $(C, P) \otimes \perp = \perp$)
- $\mathbf{0} := \perp$
- $\mathbf{1} := (\{\vec{0}\}, \emptyset)$

From these definitions we obtain that the Kleene star of a multi-linear set (C, P) can be computed in a very simple way:

$$(C, P)^* = \sum_{n=0}^{\infty} (C^n, P) = (C^*, P) = (\{\vec{0}\}, C \cup P)$$

We analyze the complexity of the operations on multi-linear sets:

- The addition $(C_1, P_1) \oplus (C_2, P_2)$ is computable in time $\mathcal{O}(|C_1| + |C_2| + |P_1| + |P_1|)$.
- The multiplication $(C_1, P_1) \otimes (C_2, P_2)$ is computable in time $\mathcal{O}(|C_1| \cdot |C_2| + |P_1| + |P_1|)$.
- The Kleene star $(C, P)^*$ is computable in time $\mathcal{O}(|C| + |P|)$.

Thus, the Kleene star of multi-linear sets is much cheaper to compute than for semilinear sets in explicit vector representation. As a further optimization, we reduce P to a minimal basis which prevents the set P from growing too large and we remove any vectors $c \in C$ which can be combined from the others as $c = c' + \sum_{p \in P} \lambda_p p$ with a $c' \in C \setminus \{c\}$.

Remark 5.14. *Similar to the GCD-approximation, multi-linear representations depend on the (syntactic) presentation of the semilinear set although finding an example is not trivial. Consider the following two representations* S_1, S_2 *of the same set* $[\![S_1]\!] = [\![S_2]\!] \subseteq \mathbb{N}^2$*:*

$$S_1 = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \mathbb{N} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\} \cup \left\{ \begin{pmatrix} 0 \\ 1 \end{pmatrix} + \mathbb{N} \cdot \begin{pmatrix} 2 \\ 0 \end{pmatrix} \right\} \cup \left\{ \begin{pmatrix} 1 \\ 1 \end{pmatrix} + \mathbb{N} \cdot \begin{pmatrix} 2 \\ 0 \end{pmatrix} \right\}$$

$$S_2 = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \mathbb{N} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\} \cup \left\{ \begin{pmatrix} 0 \\ 1 \end{pmatrix} + \mathbb{N} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}$$

The corresponding multi-linear overapproximations are[2]

$$M_1 = \left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}; \ \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 2 \\ 0 \end{pmatrix} \right)$$

$$M_2 = \left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}; \ \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right)$$

which represent different sets of vectors since $[\![M_2]\!] = \mathbb{N}^2 \neq [\![M_1]\!]$*. Thus, multi-linear sets are not a "semantic" over-approximation.*

5.3 Symbolic Representation via NDDs

The explicit vector representation of semilinear sets introduced in the last section suffers from several drawbacks that prevail even under the mentioned optimizations.

- There is no (easy, small) canonical representation (although [KT10] describe a certain normal form for semilinear sets).
- The membership problem is NP-complete, i.e. given a semilinear set S and a vector $v \in \mathbb{N}^k$ determine whether $v \in S$.
- The equivalence and inclusion problems $S_1 \overset{?}{=} S_2$, $S_1 \overset{?}{\subseteq} S_2$ are Π_p^2 complete [Huy80].

It is well-known that semilinear sets are exactly the Presburger definable subsets of $\mathbb{N}^k$, i.e. for every semilinear set S there exists a formula φ in Presburger arithmetic with free variables $x_1, \ldots, x_k$ such that

$$S = \{x \in \mathbb{N}^k : \varphi(x_1, \ldots, x_k) \equiv \text{true}\}$$

and vice versa. Note that the interesting direction is that every set that is definable in Presburger arithmetic is semilinear.

[2]We have omitted some parenthesis for readability.

Example 5.15. *Consider the set from Example 5.5.*

$$\begin{aligned} L_1 &= (1,2) + \mathbb{N}(1,2) + \mathbb{N}(2,1) \\ L_2 &= (2,3) + \mathbb{N}(1,1) + \mathbb{N}(0,2) + \mathbb{N}(2,0) \end{aligned}$$

It is straightforward to build a Presburger formula φ *defining* $S = L_1 \cup L_2$

$$\begin{aligned} \varphi(x,y) := {} & \exists \lambda, \mu \, (x = 1 + \lambda \cdot 1 + \mu \cdot 2 \wedge y = 2 + \lambda \cdot 2 + \mu \cdot 1) \\ & \vee \exists \lambda, \mu, \tau \, (x = 1 + \lambda \cdot 1 + \tau \cdot 2 \wedge y = 2 + \lambda \cdot 1 + \mu \cdot 2) \end{aligned}$$

There are well-known decision procedures for Presburger arithmetic based on (deterministic) finite automata. The idea is that natural numbers can be represented in binary, i.e. as words in $\{0,1\}^*$. This representation depends upon the bit-order (most/least significant bit first – msbf/lsbf) which is globally fixed (for example the tool MONA uses the lsbf encoding). Note that the representation is not unique, e.g. the number $6 = 2^1 + 2^2$ is 011 in binary (lsbf encoding) and is thus represented by every word in $011(0)^*$.

Analogously, one can represent vectors in $\mathbb{N}^k$ as words over the alphabet $\{0,1\}^k$ of k-dimensional binary vectors. Subsets of $\mathbb{N}^k$ then correspond to languages over $\{0,1\}^k$ and it can be shown that semilinear sets can be represented by *regular* languages over $\{0,1\}^k$. However, note that not every regular language over $\{0,1\}^k$ represents a semilinear set [Ler05]. This means that we can represent semilinear sets in a canonical way by minimal DFAs over the alphabet $\{0,1\}^k$. These minimal automata are usually called number decision diagrams (NDDs) [BC96, WB95].

Even if NDDs can be as space-consuming as our explicit vector representation, in practice they are usually small. In this sense they behave like BDDs [Bry86, And98] for representing boolean sets or relations. Moreover all operations that are rather complex for explicitly represented sets (e.g. membership/inclusion/equivalence testing or intersection) are computable in polynomial time for NDDs using the standard algorithms for DFAs.

There are several libraries (MONA, LASH) supporting the efficient computation with NDDs [HJJ+95, FL02]. In FPSOLVE we use the GENEPI framework[3] to handle NDDs. GENEPI was developed by Leroux and Point and is part of a larger collection of libraries called TAPAS [LP09]. The GENEPI framework is an abstraction layer that provides generic constructions and operations on Presburger definable sets, e.g.

- Creation of a set recognizing the solutions of a linear equation,
- Intersection and union of two sets,
- Projection of a Presburger set in $\mathbb{N}^k$ onto the subspace $\mathbb{N}^{k-1}$,
- Testing inclusion between sets.

[3]See `http://tapas.labri.fr/trac/wiki/GENEPI`.

GENEPI can be used with different representations of Presburger definable sets such as NDDs (MONA, LASH, LIRA) or Presburger formulæ (OMEGA). Each of these representations is implemented as a plugin that can be loaded dynamically. The benefit of using the generic interface provided by GENEPI is that we have the flexibility to swap the underlying representation of Presburger sets and to experiment with them. In our experiments, the most efficient representation of Presburger sets was provided by the MONA backend which we use as a default in FPSOLVE.

5.3.1 Representing Semilinear Sets via NDDs

Although it is straightforward to represent semilinear sets by NDDs, we also have to be able to compute the semiring operations $(+, \cdot, ^*)$ directly on our representation.

First note that this is clearly possible (at least in theory): there are algorithms to recover the explicit vector representation from an NDD (given that the NDD represents a semilinear set, as NDDs can represent a larger class of sets) [Ler05]. However, these algorithms are rather complex and it would defy the use of the compact NDD structure if we had to "unpack" it for each semiring operation, so we define the semiring operations on NDDs directly.

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be NDDs for the semilinear sets $S_1, S_2 \subseteq \mathbb{N}^k$. Computing the addition of the two NDDs translates to taking the union of the two NDDs, which is commonly implemented by means of the standard union construction on DFAs (followed by minimization).

$$\mathcal{A}_1 \oplus \mathcal{A}_2 := \mathcal{A}_1 \cup \mathcal{A}_2.$$

To define the multiplication, note that the $\otimes$ operation on two NDDs must satisfy

$$[\![\mathcal{A}_1 \otimes \mathcal{A}_2]\!] = S_1 \otimes S_2 = \{x + y : x \in S_1, y \in S_2\}.$$

It is easy to see that the latter set can be defined by means of a Presburger formula as

$$\{z : \exists x, y\ (z = x + y \land x \in S_1 \land y \in S_2)\}.$$

This translates directly to an algorithm for multiplying two NDDs:

1. Compute the NDD $\mathcal{A}_+$ recognizing the set $\{(x, y, z) \in \mathbb{N}^{3k} : z = x + y\}$.

2. Compute the NDDs $\mathcal{A}_1'$ recognizing $\{(x_1, x_2, x_3) \in \mathbb{N}^{3k} : x_1 \in S_1\}$ by adding new components to the vectors in $\mathcal{A}_1$. Analogously, compute the NDD $\mathcal{A}_2'$ for the set $\{(x_1, x_2, x_3) \in \mathbb{N}^{3k} : x_2 \in S_2\}$.

3. Compute the intersection $\mathcal{A}_\cap = \mathcal{A}_+ \cap \mathcal{A}_1' \cap \mathcal{A}_2'$ recognizing $\{(x, y, z) \in \mathbb{N}^{3k} : z = x + y \land x \in S_1 \land y \in S_2\}$.

4. Obtain the final result $\mathcal{A}$ by projecting the alphabet of the NDD $\mathcal{A}_\cap$ (vectors of length 3k) to its last k components.

All required operations (intersection, projection, adding components to vectors) are readily available in the GENEPI framework. Note that the projection in the last step makes the underlying automaton (temporarily) nondeterministic. This triggers a determinization step followed by minimization and thus is usually expensive. Computing the "generic sum" NDD $\mathcal{A}_+$ is also expensive since its size is exponential in k. A simple remedy is to reorder the steps in the above procedure to interleave the intersection and projection operations. For this we only compute an NDD for the three-dimensional set $\{(x, y, z) \in \mathbb{N}^3 : z = x + y\}$ and then carry out the above operations on each such triple of coordinates separately.

Further note that we could also define the $\otimes$ operation more directly if we had access to the underlying "low-level" automaton representation of the NDDs. However this would prevent us from using the GENEPI library for the implementation. Note that we use GENEPI to be able to easily switch out the underlying low-level implementations of NDDs.

Computing the Kleene star is more challenging since it is not obvious how to define the resulting set by means of a Presburger formula. To explain our algorithm, consider a semilinear set comprising two linear sets $S = c_1 P_1^* + c_2 P_2^*$. We have

$$\begin{aligned} S^* &= (1 + c_1 c_1^* P_1^*) \cdot (1 + c_2 c_2^* P_2^*) \\ &= 1 + c_1 c_1^* P_1^* + c_2 c_2^* P_2^* + (c_1 c_2) c_1^* c_2^* P_1^* P_2^* \\ &= 1 + S(c_1^* c_2^*) + S^2 (c_1^* c_2^*), \end{aligned}$$

where the last equality holds because of idempotence and since $x \in S$ implies that $x^* \subseteq S^*$. For a semilinear set comprising n linear sets with constants $\{c_1, \dots, c_n\} =: C$, we obtain by induction that

$$S^* = 1 + \left(\sum_{i=1}^{n} S^i \right) \cdot C^*.$$

Exploiting idempotence, this term can be efficiently computed by iterating

$$s_{k+1} = (1 + s_k)^2$$

until $s_{k+1} \cdot C^* = s_k \cdot C^* = S^*$, starting with $s_0 = S$. It is easy to see that $\leqslant \lceil \log n \rceil$ iterations suffice and hence the computation requires $\mathcal{O}(\log n)$ multiplications of NDDs.

The above procedure allows us to compute the Kleene star of a semilinear set represented as an NDD *if we know the set* C *of constants*. We can easily achieve this by keeping track of the set of constants C in the course of computing with the NDDs.

Thus, formally our representation turns into a pair $(C, \mathcal{A})$ and addition and multiplication are defined by

$$\begin{aligned} (C_1, \mathcal{A}_1) \oplus (C_2, \mathcal{A}_2) &:= (C_1 \cup C_2, \mathcal{A}_1 \cup \mathcal{A}_2) \\ (C_1, \mathcal{A}_1) \otimes (C_2, \mathcal{A}_2) &:= (C_1 \cdot C_2, \mathcal{A}_1 \otimes \mathcal{A}_2) \end{aligned}$$

where the product between the NDDs $\mathcal{A}_1 \otimes \mathcal{A}_2$ is computed as before and $C_1 \cdot C_2 = \{c_1 + c_2 : c_1 \in C_1, c_2 \in C_2\}$.

Further, we regard two representations $(C_1, \mathcal{A}_1)$ and $(C_2, \mathcal{A}_2)$ as equal if $\mathcal{A}_1 = \mathcal{A}_2$ (as NDDs, which are unique minimal DFAs). By disregarding the sets C_i in this equality check, our data structure yields a unique representation of semilinear sets.

5.3.2 Optimizations

At first sight, the "hybrid" representation of a semilinear set as a pair $(C, \mathcal{A})$ seems too space-consuming since the set of constants C can grow very large. However, since the set C is only used via C^* when computing the Kleene star we can instead store a much smaller set B which satisfies $B^* = C^*$. A straightforward idea would be to choose B as the minimal basis of cone(C). However, since C may contain the 0-vector which would be eliminated when computing a basis B we have to be more careful. Hence we compute a reduced B that is "almost" a basis but we keep any potential 0-vector.

Definition 5.16. *A finite set* $B = \{b_1, \ldots, b_n\} \subseteq \mathbb{N}^k$. *is called* reduced, *if no* $b \in B$ *is a combination of vectors in* $B \setminus \{b\}$ *with* positive *coefficients, i.e. there is no* $b \in B$ *such that for some* $I \subseteq [n]$ *and* $\lambda_i \in \mathbb{N} \setminus \{0\}$ *we have* $b = \sum_{i \in I} \lambda_i \cdot b_i$.

We denote by C^{red} *the (unique) reduced subset of a finite set* C.

In our optimized NDD representation, a semilinear set $S = \sum_i (c_i \cdot P_i^*)$ is represented as the pair $(C^{\text{red}}, \mathcal{A})$ with the reduced $C^{\text{red}} \subseteq C = \{c_i : i \in [n]\}$ and an NDD $\mathcal{A}$ representing the set of vectors $[\![S]\!] \subseteq \mathbb{N}^k$.

The operations on this representation are defined as before, using an additional reduction step to minimize the set B. Let $(B_1, \mathcal{A}_1)$ and $(B_2, \mathcal{A}_2)$ be representations for semilinear sets $S_1, S_2 \subseteq \mathbb{N}^k$. Addition is defined as before, followed by reduction

$$(B_1, \mathcal{A}_1) \oplus (B_2, \mathcal{A}_2) := ((B_1 \cup B_2)^{\text{red}}, \mathcal{A}_1 \cup \mathcal{A}_2).$$

Multiplication is defined analogously:

$$(B_1, \mathcal{A}_1) \otimes (B_2, \mathcal{A}_2) := ((B_1 \cdot B_2)^{\text{red}}, \mathcal{A}_1 \otimes \mathcal{A}_2).$$

Note that if $\vec{0} \in B_1$ then $B_2 \subseteq B_1 \cdot B_2$ and also $B_2 \subseteq (B_1^{\text{red}} \cdot B_2^{\text{red}})$ which would not hold if the reduction would eliminate the 0-vector from B_1.

It is easy to show that computing the addition and multiplication using reduced sets of constants preserves the invariant that $B^* = C^*$ for C the set of constants of the set, i.e. formally, we have

$$\text{cone}((B_1 \cup B_2)^{\text{red}}) = \text{cone}(B_1 \cup B_2)$$
$$\text{cone}((B_1 \cdot B_2)^{\text{red}}) = \text{cone}(B_1 \cdot B_2)$$

With this optimization, the NDD representation is quite useful for checking membership or inclusion between semilinear sets. To this end, we implemented a translation from the explicit vector representation to the NDD representation in FPsolve, which we use whenever equivalence or inclusion must be checked.

5.4 Application: A Tool for Grammar Testing

A common approach for the verification of recursive programs is to model them as pushdown systems. The set of runs of such a pushdown system is a context-free language L_1 (given e.g. as a CFG G_1). Now suppose we are given a context-free specification $L_2 = \mathcal{L}(G_2)$ and want to check whether all runs of the system conform to the specification. Note that it is a classical undecidable problem to check if $L_1 = L_2$ (or more generally, if $L_1 \subseteq L_2$) for CFLs L_1, L_2.

We can view the CFGs G_1, G_2 as algebraic systems over the language semiring $(2^{\Sigma^*}, \cup, \cdot, \emptyset, \{\varepsilon\})$, i.e. we interpret every $a \in \Sigma$ by the singleton language $\iota(a) = \{a\}$. The least solutions of these systems are then the CFLs L_1, L_2.

We now use the observation that if $L_1 \subseteq L_2$ then for every idempotent ω-continuous semiring S and every $\iota : \Sigma \to S$ the least solutions s_1, s_2 of the corresponding algebraic systems over S satisfy $s_1 \leqslant s_2$. In particular, if $L_1 = L_2$, then $s_1 = s_2$.

We can use this fact to test the in-equivalence of CFGs: to show $L_1 \neq L_2$ it suffices to exhibit an idempotent semiring S and an interpretation $\iota : \Sigma \to S$ such that $s_1 \neq s_2$ for the solutions of the associated algebraic systems.

Example 5.17. *As a very simple example consider the two grammars* G_1

$$\begin{aligned} S_1 &\to a \cdot S_1 \cdot B_1 \mid B_1 \\ B_1 &\to b \cdot B_1 \cdot X_1 \mid \varepsilon \\ X_1 &\to a \mid b \end{aligned}$$

and G_2 *(taken from the examples shipped with the tool* CFGANALYZER*)*

$$\begin{aligned} S_2 &\to a \cdot A_2 \mid b \cdot B_2 \\ A_2 &\to a \cdot C_2 \mid b \cdot A_2 \\ B_2 &\to a \cdot A_2 \mid b \cdot B_2 \\ C_2 &\to b \cdot D_2 \mid \varepsilon \\ D_2 &\to b \cdot D_2 \mid \varepsilon \end{aligned}$$

We can read the grammars as equations over the (idempotent) tropical semiring $\mathcal{T} = (\mathbb{N}_\infty, \min, +, \infty, 0)$ *by fixing some interpretation* ι *of the constants* a *and* b*, e.g.* $\iota(a) := 1$ *and* $\iota(b) := 2$*. Syntactically, we also replace "*$|$*" by the addition operation "*$\min$*" on* $\mathcal{T}$ *and "*$\cdot$*"*

by the multiplication operator "+". Finally, ε is mapped to the one-element, which is 0. Now these equations can be solved over $\mathcal{T}$ in a bottom-up fashion:

$$\begin{aligned} X_1 &= \min(1,2) = 1 \\ B_1 &= \min(2 + B_1 + X_1, 0) = 0 \\ S_1 &= \min(1 + S_1 + B_1, B_1) = 0 \end{aligned}$$

$$\begin{aligned} C_2 &= \min(2 + D_2, 0) = 0 \\ D_2 &= \min(2 + D_2, 0) = 0 \\ A_2 &= \min(1 + C_2, 2 + A_2) = \min(1, 2 + A) = 1 \\ B_2 &= \min(1 + A_2, 2 + B_2) = \min(1 + 1, 2 + B) = 2 \\ S_2 &= \min(1 + A_2, 2 + B_2) = 2 \end{aligned}$$

We can easily see that $S_1 \neq S_2$ over $\mathcal{T}$ and hence we can conclude that $\mathcal{L}(S_1) \neq \mathcal{L}(S_2)$. In fact it is easy to see here that $\varepsilon \in \mathcal{L}(S_1)$ but $\varepsilon \notin \mathcal{L}(S_2)$ which is also indicated by the fact that $S_2 = 2 \neq 0$ over $\mathcal{T}$.

In this example, we interpreted the alphabet symbols with concrete values from the (commutative) tropical semiring. In general, it depends on the chosen interpretation whether we can distinguish two grammars using this method. The "most powerful commutative distinguisher" in this sense are the commutative languages, i.e. the semiring $\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle$ (with the identity function on Σ as "interpretation"). More precisely, if two grammars are equivalent w.r.t. $\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle$ then they are equivalent w.r.t. any commutative and idempotent semiring.

We have developed a small tool that uses the above methodology to test CFGs for equivalence. The tool also constitutes a simple case study on how to use the FPSOLVE library. Currently, the tool tests CFGs by comparing the solutions to the associated algebraic system over two different idempotent semirings, one commutative the other non-commutative:

- Semilinear sets (commutative) as described in this chapter.
- Subword closures of formal languages (non-commutative), see Chapter 7.

Using the infrastructure provided by the FPSOLVE library, our implementation comprises merely 200 lines of C++. To illustrate the usefulness of the semilinear set implementations described in this chapter, we only consider this commutative abstraction in the following.

As input our tool gets two grammars and interprets them as algebraic systems over semilinear sets. It computes the solution of both systems using our explicit vector representation. The two solutions are then checked for equivalence by converting the vector representation to the unique NDD representation.

We ran the testing tool on a benchmark shipped with the tool CFGANALYZER which implements semi-decision procedures for grammar problems via SAT-solving [AHL08]. The benchmark comprises $1,892$ student solutions to problems from an introductory course on formal languages together with 40 sample solutions. Note that these grammars are rather simple with at most 3 terminal symbols (over alphabets $\{a\}$, $\{a, b\}$, $\{a, b, c\}$, and $\{0, 1\}$, respectively) and 7 non-terminals on average (the maximum being 39). We compared all student solutions with all sample answers *for a particular alphabet*, giving rise to $35,910$ pairs of grammars to be tested.

Our tool took below 20 milliseconds for each check on average while hitting the given timeout of 15 seconds for only 32 pairs. Altogether, we found that $35,031$ pairs of grammars generate different, and 841 generate equivalent languages *modulo commutativity*. For comparison, the SAT-based tool CFGANALYZER finds $35,388$ differences, which shows that the commutative approximation via semilinear sets is rather precise and a suitable tool to check (in)-equivalence of CFGs in many cases.

5.5 Conclusions

In this chapter we have presented two different representations for semilinear sets of vectors in $\mathbb{N}^{|\Sigma|}$: An explicit vector representation and a symbolic representation based on automata (NDDs).

We have described how to implement the explicit vector representation in a space efficient way and have described over-approximations that can be used to further reduce their size.

Semilinear sets are (isomorphic to) the rational elements of the semiring $\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle$. Since Newton's method converges in $n + 1$ steps over all commutative and idempotent semirings [EKL07b], semilinear sets can be used to represent solutions of algebraic systems over the "most general" commutative and idempotent semiring $\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle$. Equivalently, such a solution describes the Parikh image of the associated context-free grammar and we have described how to use this fact to test CFGs for (in-)equivalence modulo commutativity.

Parikh Image Although our explicit vector representation of semilinear sets is useful for computing Parikh images of small grammars this representation does not scale to larger grammars. In some sense this is inevitable for this representation since the Parikh image of a CFG can contain exponentially many linear sets (see the seminal paper [KT10] of Kopczyński and To for a detailed analysis).

Despite the optimizations described in Section 5.3.2, the practical performance of our NDD representation is still rather poor as well. Currently, the main bottleneck is the multiplication of two NDDs (and hence the Kleene star as well). Thus, future work on this representation should mainly focus on improving the complexity of the multiplication operation.

Verma, Seidl, and Schwentick present an efficient (linear-time) construction of a Presburger formula defining the Parikh image of a CFG [VSS05]. Recently, Brugger studied in his Bachelor's thesis [Bru14] whether this can be used to check CFGs for equivalence by means of an SMT-solver but found that the resulting (quantified) formula is usually too challenging to solve for current solvers, even for small grammars.

6 Application I: Provenance for Datalog

Roughly speaking, *database provenance* is additional information attached to base facts in a database that is propagated through query evaluation. The provenance of the query result then describes how a particular fact was derived from the base facts.

In this chapter we show how to efficiently represent and compute provenance information for Datalog programs. We start by introducing the basics of Datalog and the semiring framework proposed by Green et al. [GKT07] in Section 6.1.

In Section 6.2 we present an application of our results for Newton's method to Datalog provenance. In particular, we propose to view recursive Datalog programs directly as algebraic systems which separates the problems of *defining* and *computing* provenance more clearly (cf. Section 6.2.1). We also show that the results of [DMRT14] on *absorptive* semirings are already implied by [EKL08] since this class coincides with the 1-bounded semirings defined there.

Next, in Section 6.2.2 we propose $\mathbb{N}_k\langle\langle\Sigma^{\oplus}\rangle\rangle$ as a general semiring for representing Datalog provenance and show how to represent its elements concisely by shared regular expressions. We also show that we can compute these expressions in polynomial time via Newton's method (cf. Theorem 6.8).

Finally, we present our main result in Section 6.2.4 where we describe how to specialize our shared regular expressions efficiently over the so called why-semiring to yield concise expressions without Kleene stars. This refutes a claim made in [DMRT14] and demonstrates that a special treatment of the why-semiring as in [DMRT14] is not necessary.

6.1 Introduction to Datalog and Provenance

The term *provenance*, meaning “origin” or “linage”, has become a buzzword in the field of data and systems research during the last 10 years. The rather fuzzy nature of the term is evident from its use for various problems in different areas of computer science:

- Authorship: Identifying the author of a piece of text or program code. This author can be a human or (e.g. in the case of machine code) a program like a compiler [Ros11].
- Reproducibility and Traceability: Tracing the steps executed during a scientific workflow, e.g. in a computational experiment. Record all facts about the environment needed to reproduce the results, e.g. machine or operating system configurations, the origin of the data used, the nature of any data cleaning processes, etc. [FKSS08, FBS12].
- Databases: Explaining/Tracing how the results of a database query were derived. Such information is helpful for
  1. the database user to find flaws in her queries,
  2. the database engineer to debug the design, and
  3. the database system to improve its efficiency, e.g. by enabling efficient propagation of deletions or updates without needing to recompute all views or query results.

Here, we only consider data provenance for Datalog programs and focus on the formally well-defined semiring framework proposed by Green et. al [GKT07, CCT09].

6.1.1 Datalog

Datalog is a deductive programming language (a strict subset of Prolog) and is often used as a query language in deductive databases. Datalog has experienced a remarkable revival in the last 10 years [HGL11], with many applications, most prominently in program analysis [WACL05, SB11]. We refer to [AHV95] for a thorough introduction to Datalog and only give a brief overview here.

A Datalog program P consists of a finite set of rules of the form

$$R_0(x_0) :\!- R_1(x_1), \ldots, R_k(x_k).$$

where R_i denote relation symbols and if R_i is of arity n then x_i is a vector of n variables. From a logical point of view, these rules are Horn-clauses

$$R_1(x_1) \wedge \cdots \wedge R_k(x_k) \to R_0(x_0),$$

with R_0 the *head* and $R_1, \ldots, R_k$ the *body* of the clause.

A relation symbol R is called *extensional* if it never occurs as a head. The set of all extensional symbols is called the extensional data base (EDB). Relation symbols that occur as heads of rules are called *intensional* and taken together they form the intensional data base (IDB).

An *instance* I of the EDB of a Datalog program is a finite set of ground formulæ of the form $R(a_1, \ldots, a_n)$ with an extensional relation symbol R and constants a_i. In logic programming, the elements of the instance are also called (base) *facts* and viewed as part of the program.

Example 6.1. *The following Datalog program defines the intensional ancestor relation* Anc *using the extensional parent relation* Par*:*

$$\begin{aligned} \mathsf{Anc}(x,y) &:\!-\ \mathsf{Par}(x,y). \\ \mathsf{Anc}(x,y) &:\!-\ \mathsf{Anc}(x,z), \mathsf{Anc}(z,y). \end{aligned}$$

A possible instance over $\{\mathsf{Par}\}$ *is*

$$\begin{array}{rr} \mathsf{Par}(\mathit{Lindemann}, \mathit{Hilbert}) & \mathsf{Par}(\mathit{Lindemann}, \mathit{Minkowski}) \\ \mathsf{Par}(\mathit{Lindemann}, \mathit{Perron}) & \mathsf{Par}(\mathit{Minkowski}, \mathit{Caratheodory}) \\ \mathsf{Par}(\mathit{Hilbert}, \mathit{Curry}) & \mathsf{Par}(\mathit{Hilbert}, \mathit{Courant}) \end{array}$$

from which we could conclude that Lindemann is an ancestor of Courant, or

$$\mathsf{Anc}(\mathit{Lindemann}, \mathit{Courant})$$

In this example we have relied on intuition to give a semantics to a Datalog program. A formal semantics has to specify how to interpret the relation symbols in the IDB (i.e. the relation they should define) given a concrete instance over the EDB. There are several possibilities to give a semantics to a Datalog program which are all equivalent for our setting[1] (see [AHV95]):

- Least-model-semantics: given an instance I, the semantics of the program P is the minimum (w.r.t. inclusion) model of the set of clauses in P that contains I.
- Least-fixpoint semantics: viewing the program P as a (monotone) operator F_P mapping instances to instances (the *immediate consequence operator*). The semantics of P is the least fixpoint of F_P obtained by computing the sequence $I, F_P(I), F_P^2(I), \ldots$ until it stabilizes.

[1]If we consider an extension of Datalog with logical negation, it is more difficult to assign a well-defined meaning to programs.

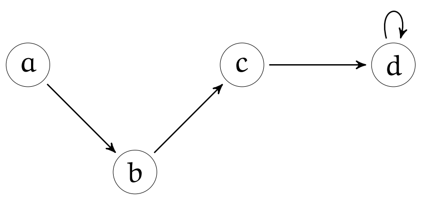
- Proof-theoretic semantics: A relation $R(x)$ holds, if there exists a proof-tree for the fact $R(x)$ using the program as inference rules and the facts in I as axioms.

Example 6.2. *As a more general (and abstract) version of the previous example we consider the problem of computing the transitive closure* T *of a (directed) graph specified by an extensional edge relation* E. T *can be specified by means of the following Datalog program* P

$$\begin{aligned} &T(x,y) \mathbin{:\!-} E(x,y).\\ &T(x,y) \mathbin{:\!-} T(x,z), T(z,y). \end{aligned}$$

A possible instance is given by a set of edges, describing the graph below.

$$I = \{E(a,b), E(b,c), E(c,d), E(d,d)\}.$$



The resulting transitive closure is then given by

$$\mathcal{C} = \{T(a,b), T(b,c), T(c,d), T(d,d), T(a,c), T(a,d), T(b,d)\}.$$

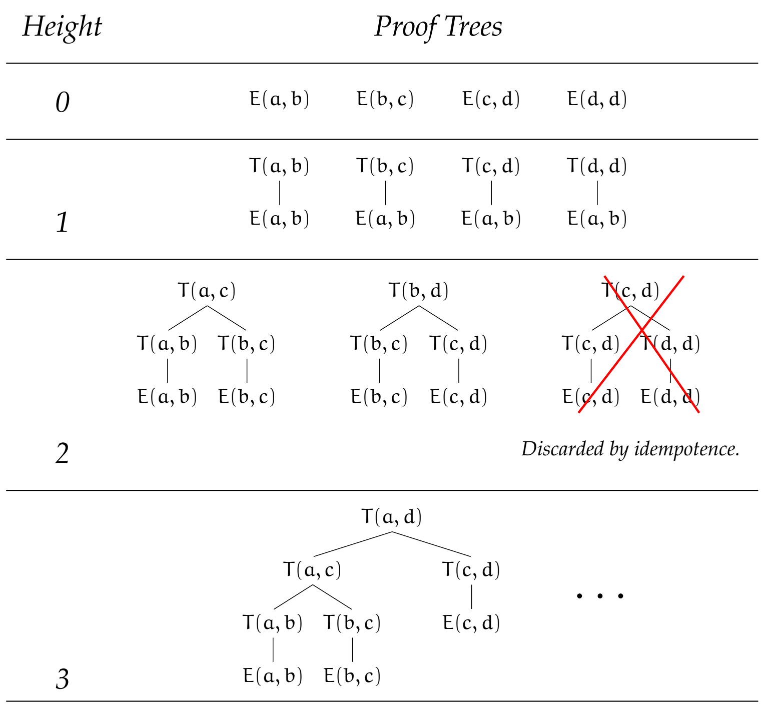
and the unique minimal model of the program P *(viewed as clauses) is* $\mathcal{M} = I \cup \mathcal{C}$. *Note that*

$$\mathcal{M}' = I \cup \mathcal{C} \cup \{T(b,a), T(a,a), T(b,b)\}$$

also constitutes a valid model of P, *but is not minimal. We can compute the minimal model* $\mathcal{M}$ *by iterating the immediate consequence operator* F_P *associated with our program until we reach a fixpoint:*

$$\begin{aligned} F_P^1(I) &= I \cup \{T(a,b), T(b,c), T(c,d), T(d,d)\}\\ F_P^2(I) &= F_P^1(I) \cup \{T(a,c), T(b,d)\}\\ F_P^3(I) &= F_P^2(I) \cup \{T(a,d)\}\\ F_P^4(I) &= F_P^3(I) \end{aligned}$$

Under the proof-theoretic view, the semantics of the program comprises all derivable facts. These can be obtained by considering the set of all proof trees. Since we are interested in sets of facts we can discard proof trees of facts that are already known. We can systematically enumerate all proof trees by increasing height, obtaining in each "level" exactly the facts that are discovered by fixpoint iteration.


Height
Proof Trees
0
E(a, b) E(b, c) E(c, d) E(d, d)
1
T(a, b) T(b, c) T(c, d) T(d, d)
E(a, b) E(a, b) E(a, b) E(a, b)
2
T(a, c) T(a, b) T(b, c) E(a, b) E(b, c)
T(b, d) T(b, c) T(c, d) E(b, c) E(c, d)
T(c, d) T(c, d) T(d, d) E(c, d) E(d, d)
Discarded by idempotence.
3
T(a, d) T(a, c) T(c, d) T(a, b) T(b, c) E(c, d) E(a, b) E(b, c)
. . .

6.1.2 Green's Semiring Framework for Provenance

Here we review the semiring framework for provenance proposed by Green et al. [GKT07] by means of examples. We also give a survey of the provenance hierarchy defined by Green [Gre09b, Gre09a].

Consider the Datalog program below computing the relation x using a base relation r depicted as the table on the left (the underscores "_" represent "don't care" variables). The result relation x (depicted on the right) is obtained by the following procedure:

1. Join two tuples t_1, t_2 from r if
   a) the second component of t_1 is "b" and
   b) the first (A) or the last (C) component of t_1 and t_2 are equal,
2. then the respective other components, i.e. $t_1(A), t_2(C)$ or $t_1(C), t_2(A)$ form the result.

A	B	C
a	b	a
a	b	c
b	c	c
b	a	a

$$
\begin{aligned}
x(X, Y) &\ :\!-\, r(X, b, Z), r(Y, \_, Z).\\
x(X, Y) &\ :\!-\, r(Z, b, X), r(Z, \_, Y).
\end{aligned}
$$

X	Y
a	a
a	b
a	c
c	a
c	c

To explain the existence of the result $x(a, c)$ we can annotate the original data with abstract variables p, q, r, s representing boolean values. The annotation of result obtained by joining t_1 and t_2 is then obtained by performing the "And" of the two annotations. Similarly if a result can be obtained via an alternative derivation we take the "Or" of the annotations. Hence, our annotations are positive (i.e. negation-free) Boolean formulæ.

A	B	C	Tag
a	b	a	p
a	b	c	q
b	c	c	r
b	a	a	s

X	Y	Tag
a	a	$(p \wedge p) \vee (q \wedge q) = p \vee q$
a	b	$(p \wedge s) \vee (q \wedge r)$
a	c	$p \wedge q$
c	a	$p \wedge q$
c	c	$s \wedge s = s$

We can then verify that $x(a, c)$ is a result since it can be derived using the tuples tagged by p and q

$$x(a, c) :\!- \, r(a, b, a), r(a, b, c).$$

Furthermore, we can study the effect of a deletion in r on the result x: For example deleting the tuple (a, b, c) from r corresponds to setting $q = \text{FALSE}$ in our annotation which tells us that the tuples (a, c) and (c, a) would be missing in x after the deletion.

In contrast to Datalog's set-semantics, other systems (employing SQL as query language) usually exhibit *bag-semantics*, i.e. the result of a query is a multiset of facts. We can model bag-semantics in Datalog as provenance by tagging facts with multiplicities in $\mathbb{N}$, multiplying tags when performing a join and adding tags of equal tuples in the result.

Assuming multiplicities of 1 for all facts in r we obtain the relation x with respective multiplicities:

X	Y	Mult.
a	a	3
a	b	2
a	c	1
c	a	1
c	c	1

The Boolean annotations from above do not explain the multiplicity 3 of the tuple (a, a). To get a more detailed trace of the derivations we interpret p, q, r, s as elements of some alphabet Σ and view the provenance annotations as elements of the semiring of polynomials $\mathbb{N}\langle\Sigma^{\oplus}\rangle$ in commuting variables Σ. Again, we multiply annotations when a tuple is obtained as result of a join and add annotations when a tuple is the result of a union or projection operation. Computing the annotated relation x in this way we obtain:

A	C	Tag
a	a	$2p^2 + q^2$
a	b	$ps + qr$
a	c	pq
c	a	pq
c	c	s^2

These annotations tell us that the fact $x(a, a)$ can be derived in three different ways and records the input tuples participating in the derivation. More precisely, we can obtain $x(a, a)$ as follows:

1. Using the fact $r(a, b, c)$ twice in the first rule with the unification $X = Y = a$ and $Z = c$.
$$x(a, a) :\text{-}\, r(X, b, Z), r(Y, \_, Z)$$
2. Using the fact $r(a, b, a)$ twice in the first rule with the unification $X = Y = Z = a$.
$$x(a, a) :\text{-}\, r(X, b, Z), r(Y, \_, Z)$$
3. Using the fact $r(a, b, a)$ twice in the second rule with the unification $X = Y = Z = a$.
$$x(a, a) :\text{-}\, r(Z, b, X), r(Z, \_, Y)$$

Furthermore, we can use the provenance polynomials from $\mathbb{N}\langle\Sigma^{\oplus}\rangle$ to explore the effect of different multiplicities in our input relation r:

A	B	C	Mult.
a	b	a	$p = 1$
a	b	c	$q = 2$
b	c	c	$r = 5$
b	a	a	$s = 3$

$$\overset{x}{\rightsquigarrow}$$

A	C	Mult.
a	a	$2 + 4 = 6$
a	b	$3 + 10 = 13$
a	c	2
c	a	2
c	c	9

For recursive Datalog programs like the transitive closure program

$$T(x, y) :\text{-}\, E(x, y).$$
$$T(x, y) :\text{-}\, T(x, z), T(z, y).$$

the provenance of a result tuple can be an infinite expression (i.e. a power series in $\mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle$):

X	Y	Tag
a	b	p
c	c	r
b	c	q

$\overset{T}{\rightsquigarrow}$

X	Y	Tag
a	b	p
c	c	$r + r^2 + 2r^3 + 5r^4 + \dots$
b	c	$q + qr + qr^2 + 2qr^3 + \dots$
a	c	$pq + pqr + pqr^2 + 2pqr^3 + \dots$

We can observe that the power series representing the provenance of (c, c) is (almost[2]) the generating function of the binary trees. Every proof tree of our transitive closure program is a binary tree and the fact (c, c) can be derived using the base fact (c, c) exactly k times for any $k \in \mathbb{N} \setminus \{0\}$ giving rise to a binary proof tree with k leaves. As a result there are $C_{k-1} = \frac{1}{k}\binom{2(k-1)}{(k-1)}$ different proof trees for every $k \geqslant 1$. Since the provenance information is infinite, evaluating the query under the fixpoint semantics does not terminate.

Green et al. [GKT07] notice that provenance can be abstractly specified as the least solution of an algebraic system (over $\mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle$). For example the provenance of the tuple (c, c) in the above example is given by the least solution of

$$x = x^2 + r.$$

They only solve these algebraic systems if the solution is a polynomial in $\mathbb{N}_\infty\langle\Sigma^\oplus\rangle$ (i.e. a finite sum) and their algorithm enumerates derivation trees by height and hence is equivalent to standard fixpoint iteration. If the procedure detects a "pumpable" subtree (i.e. a fact that occurs twice along a path from the root) it aborts the evaluation (since the solution is an infinite sum in this case).

Survey of the Provenance Hierarchy As observed by Green [Gre09b] the different semirings used to encode provenance information can be organized in a hierarchy (i.e. a lattice) where the relation $A \rightarrow B$ between semirings A and B holds if there exists a surjective homomorphism $h : A \rightarrow B$. A dashed arrow from A to B indicates that every interpretation $\iota : \Sigma \rightarrow B$ extends uniquely to a homomorphism $h : A \rightarrow B$. Note that all semirings describing provenance are commutative ω-continuous semirings since joins in Datalog (and relational algebra) are commutative.

We visualize this lattice below as follows. The identities labeling an edge from semiring A to B denote how to obtain B from A by assuming the respective axiom. As a convention we assume a, b to be alphabet symbols in Σ and x to be semiring elements. For example the idempotence axiom for addition ($1 + 1 = 1$) could be equivalently expressed as $x + x = x$, but $a \cdot a = a$ does not specify the idempotence of multiplication.

[2]The first term of the generating function $C(r)$ is r^0, so the provenance is given by $r \cdot C(r)$.

Indeed, we obtain $(p+q)\cdot(p+q) = p + pq + q$ assuming $a \cdot a = a$ (for all $a \in \Sigma$) which is different from $(p+q)\cdot(p+q) = p + q$ assuming idempotent multiplication.

$$\begin{tikzcd}
 & & \mathbb{N}_\infty & & \mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle \arrow[ll, dashed] \arrow[dl, "1+1=1"'] \arrow[dr, "a\cdot a=a"] & \\
 & \mathcal{T}_\mathbb{Q} & & \mathbb{B}\langle\langle\Sigma^\oplus\rangle\rangle \arrow[ll, dashed] \arrow[dl, "1+x=1"'] \arrow[dr, "a\cdot a=a"] & & \mathsf{Trio}(\Sigma) \arrow[dl, "1+1=1"] \\
\mathcal{T}_\mathbb{N} & & \mathsf{Sorp}(\Sigma) \arrow[ll, dashed] \arrow[dr, "a\cdot a=a"'] & & \mathsf{Why}(\Sigma) \arrow[dl, "1+x=1"'] \arrow[dr, "ab=a+b"] & \\
 & & & \mathsf{PosBool}(\Sigma) \arrow[dr, dashed] & & \mathsf{Lin}(\Sigma) \arrow[dl, dashed] \\
 & & & & \mathbb{B} &
\end{tikzcd}$$

Green et al. [GKT07, Gre09b] specify several semirings that describe interesting provenance information.

- $\mathbb{N}_\infty, \mathbb{B}$: The extended natural numbers and the Boolean semiring, modeling bag- and set-semantics, respectively.
- $\mathcal{T}_\mathbb{Q}, \mathcal{T}_\mathbb{N}$: The tropical (min-plus) semirings over $\mathbb{Q}$ and $\mathbb{N}$.
- The semiring $\mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle$ of formal power series in commuting variables Σ is the commutative ω-continuous semiring freely generated by the elements of Σ. As such it is the "most general" commutative ω-continuous semiring.
- The semiring $\mathbb{B}\langle\langle\Sigma^\oplus\rangle\rangle$ is obtained from $\mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle$ modulo the idempotence axiom $1+1=1$. Intuitively, this means we "collapse (non-zero) coefficients to 1". As remarked in Section 5.1 the elements of $\mathbb{B}\langle\langle\Sigma^\oplus\rangle\rangle$ can be viewed as the commutative images of arbitrary formal languages.
- The $\mathsf{Why}(\Sigma)$ semiring is $\mathbb{B}\langle\langle\Sigma^\oplus\rangle\rangle$ modulo the identity $a \cdot a = a$ for all $a \in \Sigma$, hence the monomials in a power series in $\mathsf{Why}(\Sigma)$ can be identified with subsets of Σ and power series in $\mathsf{Why}(\Sigma)$ are polynomials. Therefore $\mathsf{Why}(\Sigma)$ is (isomorphic to) the semiring $(2^{2^\Sigma}, \cup, \Cup, \emptyset, \{\emptyset\})$ with $\mathcal{A} \Cup \mathcal{B} := \{A \cup B : A \in \mathcal{A}, B \in \mathcal{B}\}$ for $\mathcal{A}, \mathcal{B} \subseteq \Sigma$. This notion of Why-provenance was originally introduced by [BKWC01].
- The *linage* semiring $\mathsf{Lin}(\Sigma)$ (after [CWW00]) is obtained from $\mathsf{Why}(\Sigma)$ modulo the identity $ab = a + b$ for $a, b \in \Sigma$. Intuitively we obtain an element of $\mathsf{Lin}(\Sigma)$ from an element of $\mathsf{Why}(\Sigma)$ (represented as a set-of-sets) by "flattening" the set-of-sets

representation, e.g. $\{\{p, q\}, \{p\}\} = \{p, q, p\} = \{p, q\}$. $\mathsf{Lin}(\Sigma)$ is isomorphic to the semiring $(2^\Sigma \cup \{\bot\}, \cup, \cup, \emptyset, \bot)$ with $A \cup \bot := A$. An annotation in $\mathsf{Lin}(\Sigma)$ describes the set of base facts contributing to the derivation of a result tuple.

- The $\mathsf{PosBool}(\Sigma)$ semiring of positive Boolean formulæ can be either defined as $\mathsf{Why}(\Sigma)$ modulo $1 + x = 1$ or as the set of negation-free Boolean expressions (terms using variables and the operators "$\wedge$" and "$\vee$") where two expressions are identified if they yield the same truth value (cf. [Gre09a]). This is a consequence of the fact that the axiom $1 + x = 1$ is equivalent to the absorption axiom $x + xy = x$ from Boolean algebra. More explicitly, $\mathsf{PosBool}(\Sigma)$ is (isomorphic to) the semiring $(2^{2^\Sigma}, \cup, \uplus, \emptyset, \{\emptyset\})$ such that for each $\mathcal{S} \in (2^{2^\Sigma}$ we have $A, B \in \mathcal{S} \Rightarrow A \not\subseteq B \wedge B \not\subseteq A$.

- The $\mathsf{Trio}(\Sigma)$ semiring is modeled after [STW08] and comprises the power series from $\mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle$ modulo the identities $a \cdot a = a$ for all $a \in \Sigma$. Informally, these series are obtained by "dropping exponents" from series in $\mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle$. As a result, all series in $\mathsf{Trio}(\Sigma)$ are polynomials. However, $\mathsf{Trio}(\Sigma)$ itself is infinite and also contains infinite ascending chains since we can embed $\mathbb{N}$ into it, e.g. $1 \cdot \varepsilon \leqslant 2 \cdot \varepsilon \leqslant 3 \cdot \varepsilon \leqslant \ldots$. [STW08] use the $\mathsf{Trio}(\Sigma)$ provenance for evaluating queries in probabilistic databases, specifically the Trio-system (hence the name).

- The *absorptive* semiring (freely generated by Σ) $\mathsf{Sorp}(\Sigma)$ [DMRT14] is obtained from $\mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle$ modulo the absorption axiom $x + xy = x$ for all $x, y \in \mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle$.

6.2 Regular Expressions for Provenance

Note that although Green et al. [GKT07] mention the semirings $\mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle$ and $\mathbb{B}\langle\langle\Sigma^\oplus\rangle\rangle$ they do not use them to represent the provenance of recursive queries if it is an infinite sum. More recently, Deutch et al. [DMRT14] argue that there is no general finite representation that can be specialized to $\mathsf{Why}(\Sigma)$ and therefore propose different methods for the semirings $\mathsf{Sorp}(\Sigma)$ and $\mathsf{Why}(\Sigma)$. More precisely, Theorem 3 in [DMRT14] claims that it is not possible to represent the most general provenance $s \in \mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle$ of some IDB-fact as a finite expression (i.e. a polynomial $p \in \mathbb{N}\langle\Sigma^\oplus\rangle$) such that for any interpretation $\iota : \Sigma \to \mathsf{Why}(\Sigma)$ we have

$$h(s) = h(p)$$

for the unique extension $h : \mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle \to \mathsf{Why}(\Sigma)$.

In this section we show that a special treatment of semirings such as $\mathsf{Why}(\Sigma)$ or $\mathsf{Sorp}(\Sigma)$ is not necessary and that we can effectively use $\mathbb{B}\langle\langle\Sigma^\oplus\rangle\rangle$ or even the more general k-collapsed semiring $\mathbb{N}_k\langle\langle\Sigma^\oplus\rangle\rangle$ as provenance representations. More precisely, we make the following contributions:

- Given a Datalog program, we show how to compute shared regular expressions representing the provenance of IDB facts over $\mathbb{N}_k\langle\langle\Sigma^\oplus\rangle\rangle$ in polynomial time in the number of IDB facts (Theorem 6.8).
- We show that the semiring $\mathsf{Sorp}(\Sigma)$ is 1-bounded and hence well-known solution methods for algebraic systems can be applied [EKL08].
- We show how to eliminate the Kleene stars from the representation over the semiring $\mathsf{Why}(\Sigma)$. The resulting shared expressions only involve addition and multiplication and grow at most by a factor of $\log|\Sigma|$ in size. This shows that the claim in [DMRT14] can only be true for infinite alphabets Σ. Note that since the number of EDB facts in a database is finite, it suffices to consider finite Σ for provenance analyses.

6.2.1 Datalog Programs as Algebraic Systems

Here we propose a less operational view on Datalog provenance that does not mix the issues of specification and computation of provenance: In our view, Datalog programs *are* algebraic systems and their unique least *solution* in $\mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle$ can be interpreted over different semirings, yielding different notions of provenance. This least solution can be *computed* or *approximated* using different algorithms, of which fixpoint iteration is only one possible choice. Depending on the semiring, other algorithms (e.g. based on Newton's method) can be a better choice.

We view a Datalog program over a concrete instance I directly as a (very large) algebraic system over $\mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle$:

- The alphabet Σ is given by $\Sigma := I$.
- The set of variables $\mathcal{X}$ of the system comprises all the intensional relation symbols $R(x)$ with the vector x instantiated by all combination of constants appearing in I.
- The defining equation of a variable $R_0(x_0)$ is obtained by summing over all rules in the program P having R_0 as the head:

$$R_0(x_0) = \sum_{\substack{r\in P \\ r=R_0(x_0)\text{ :- }R_1(x_1),\ldots,R_k(x_k)}} \prod_{i\in[k]} R_i(x_i)$$

Example 6.3. *The transitive closure program*

$$T(x,y)\text{ :- }E(x,y).$$
$$T(x,y)\text{ :- }T(x,z),T(z,y).$$

on the instance $I = \{E(a,b), E(b,c), E(c,c)\}$ *gives rise to the following algebraic system of* 9 *equations in the variables* $\mathcal{X} = \{T(x,y) : x,y \in \{a,b,c\}\}$ *with* $\Sigma = \{E(a,b), E(b,c), E(c,c)\}$*.*

$$\begin{aligned}
T(a,a) &= E(a,a) + T(a,a)\cdot T(a,a) + T(a,b)\cdot T(b,a) + T(a,c)\cdot T(c,a)\\
T(a,b) &= E(a,b) + T(a,a)\cdot T(a,b) + T(a,b)\cdot T(b,b) + T(a,c)\cdot T(c,b)\\
T(a,c) &= E(a,c) + T(a,a)\cdot T(a,c) + T(a,b)\cdot T(b,c) + T(a,c)\cdot T(c,c)\\
&\vdots\\
T(c,c) &= E(c,c) + T(c,a)\cdot T(a,c) + T(c,b)\cdot T(b,c) + T(c,c)\cdot T(c,c)
\end{aligned}$$

Note that we do not advise to write out this algebraic system explicitly for actually computing the solution. We only argue that it is helpful to view a Datalog program *implicitly* as such a system.

With this interpretation, the set-semantics of a Datalog program corresponds to interpreting the associated algebraic system over the Boolean semiring $(\mathbb{B}, \vee, \wedge, 0, 1)$ by setting $R(c) = \text{TRUE}$ for all $R(c) \in I$. The semantics of the Datalog program is then given by all facts $R(c)$ such that $R(c) = \text{TRUE}$ in the least solution of the system. Analogously, we obtain bag-semantics by interpreting the system over the (extended) natural numbers $(\mathbb{N}_\infty, +, \cdot, 0, 1)$.

Example 6.4 (continued)**.** *The previous system can be simplified by removing all variables which are zero in the solution. This corresponds to determining the productive nonterminals in the associated CFG which are given by* $\{T(a,b), T(a,c), T(b,c), T(c,c)\}$*. Thus the system simplifies to*

$$\begin{aligned}
T(a,b) &= E(a,b) + T(a,c)\cdot T(c,b)\\
T(a,c) &= T(a,b)\cdot T(b,c) + T(a,c)\cdot T(c,c)\\
T(b,c) &= E(b,c) + T(b,c)\cdot T(c,c)\\
T(c,c) &= E(c,c) + T(c,c)\cdot T(c,c)
\end{aligned}$$

Over the Boolean semiring with $E(a,b) = E(b,c) = E(c,c) = \text{TRUE}$ *we would get* $T(a,b) = T(a,c) = T(b,c) = T(c,c) = \text{TRUE}$ *as the least solution which gives us the set-semantics of the Datalog program.*

Similarly, over $(\mathbb{N}_\infty, +, \cdot, 0, 1)$ *with* $E(a,b) = E(b,c) = E(c,c) = 1$ *we obtain* $T(a,b) = T(a,c) = T(b,c) = T(c,c) = \infty$ *which corresponds to the bag-semantics of the program.*

6.2.2 Extending the Provenance Hierarchy

Consider again the application of provenance to deletion propagation mentioned in Section 6.1.2. There, we want to decide if a previously computed fact is still derivable after certain base facts have been deleted from the EDB. This amounts to *evaluating* the computed provenance annotation by interpreting every variable in Σ by a Boolean value $\iota : \Sigma \to \mathbb{B}$ denoting its presence or absence.

Another application mentioned in [Gre09a] arises when integrating data from different sources. In the simplest case one can model trust as binary values, yielding the Boolean semiring. [Gre09a] also briefly mentions a more refined trust model based on distrust scores (e.g. numbers in $\mathbb{N}$) assigned to facts. In the course of a derivation distrust then adds up when joining facts. The total distrust assigned to a result is given by the minimum distrust over all derivations yielding it. This model of trust corresponds exactly to using the tropical semiring as provenance annotations.

Yet a different notion of trust can be formalized as follows: A user may assign trust values (e.g. numbers in $\mathbb{Q}$) to different sources and query results are tagged with the maximum trust of any derivation, where the trust of a particular derivation is the minimum trust over all facts involved. Formally, this amounts to provenance interpreted over the semiring $(\mathbb{Q} \cup \{\pm\infty\}, \max, \min, -\infty, +\infty)$. A similar approach to model the confidentiality of facts is the (finite) *security* semiring introduced by [ADT11a] .

In all of these applications we eventually interpret the abstract provenance tags in Σ by values from an *idempotent* semiring. Recomputing provenance even for "simple" semirings (like the Boolean semiring) may be costly if it has to be done often. Instead it can be beneficial to compute provenance information only once over the "most-general" commutative idempotent semiring $\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle$, store it in a space-efficient way, and later *evaluate* the stored representation over different specialized semirings.

Besides the motivation from applications there are also theoretical reasons for considering some form of idempotence as the following proposition shows. Note that a similar observation was made by Petre [Pet99].

Proposition 6.5 (Algebraic $\neq$ rational over $\mathbb{N}_\infty\langle\langle\Sigma^{\oplus}\rangle\rangle$)**.** *Let* r *be a solution to an algebraic system over* $\mathbb{N}_\infty\langle\langle\Sigma^{\oplus}\rangle\rangle$*. Then* r *is not (in general) a rational series, i.e. we cannot describe* r *by a regular expression.*

Proof. Consider the algebraic system $X = X^2 + c$. If we could represent its least solution via a regular expression r then under the homomorphism $h(c) = \frac{1}{8}$ we would obtain a rational number $h(r)$. However, the least solution to $X = X^2 + \frac{1}{8}$ over $\mathbb{R}$ can also be calculated directly as $X = 4 \cdot \left(1 - \sqrt{\frac{1}{2}}\right)$ which is not rational since $\sqrt{2}$ is irrational. □

In Chapter 5 we have already discussed the representation of rational sets in $\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle$ as semilinear sets using either an explicit vector representation or a symbolic NDD representation. However, if we are only interested in specializing the provenance information by means of an evaluation induced by $\iota : \Sigma \to S$ into some idempotent semiring S, these representations of $\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle$ are too space-consuming. Instead we can represent the elements in $\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle$ via regular expressions that can be concisely represented by sharing common subexpressions. This is the same shared-term representation we use in FPSOLVE as described in Section 4.2.3.

Example 6.6. *The following picture shows the shared representation of the expression* $a^*b(ca^*b + d)^*$*:*

$$\begin{tikzcd}
 & & & \cdot \arrow[dr] \arrow[ddddl] & \\
 & & & & * \arrow[d] \\
 & & & & + \arrow[dl] \arrow[dddd] \\
 & & & \cdot \arrow[dl] \arrow[dddlll, bend right] & \\
 & & \cdot \arrow[dl] \arrow[dd] & & \\
 & * \arrow[d] & & & \\
c & a & b & & d
\end{tikzcd}$$

Recall our result that Newton's method over commutative k-collapsed semirings converges in finite time (Proposition 3.38). Since all Newton approximations are rational in $\mathbb{N}_k\langle\langle\Sigma^\oplus\rangle\rangle$ we can also represent elements from this general semiring finitely by means of regular expressions. We therefore extend Green's provenance hierarchy by the semiring $\mathbb{N}_k\langle\langle\Sigma^\oplus\rangle\rangle$ which can be obtained from $\mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle$ modulo the generalized idempotence axiom $1 + k = k$ (informally, we "collapse all coefficients larger than k to k"). The so extended hierarchy is shown in Figure 6.1.

Every element of $\mathbb{B}\langle\langle\Sigma^\oplus\rangle\rangle$ can be specialized to a value in the (idempotent) tropical semiring $(\mathbb{Q} \cup \{\pm\infty\}, \min, +, \infty, 0)$ by interpreting the alphabet symbols in Σ. Similarly, we can specialize $\mathbb{N}_k\langle\langle\Sigma^\oplus\rangle\rangle$ to the non-idempotent k-tropical semiring $\mathcal{T}_{k,\mathbb{N}}$ (cf. Example 3.37) that is used to model k-shortest path analyses. Note that $\mathbb{B}\langle\langle\Sigma^\oplus\rangle\rangle$ is not general enough to allow such specialization.

To further illuminate the provenance hierarchy, we show that the class of "absorptive" semirings (e.g. $\mathsf{Sorp}(\Sigma)$) introduced in [DMRT14] is already well-known from [EKL08] as the 1-bounded semirings.

A semiring is called 1-bounded, if $x \leqslant 1$ holds (or equivalently, $x+1 = 1$). In [DMRT14] *absorptive semirings* are defined as semirings[3] which satisfy the identity

$$x + xy = x \quad \text{for all} x, y \in S$$

Every absorptive semiring is idempotent, since it satisfies $x + x \cdot 1 = x$. We can further prove that the notions of 1-bounded and absorptive semirings coincide.

Proposition 6.7. *A semiring S is* 1*-bounded if and only if it is absorptive.*

[3]In [DMRT14] also commutativity is assumed. This is not necessary to show Proposition 6.7.

$$\begin{tikzcd}
 & & & \mathbb{N}_\infty & & \mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle \arrow[ll, dashed] \arrow[dl, "1+k=k"'] \arrow[dr, "a\cdot a=a"] & \\
 & & \mathcal{T}_{k,\mathbb{N}} & & \mathbb{N}_k\langle\langle\Sigma^\oplus\rangle\rangle \arrow[ll, dashed] \arrow[dl, "1+1=1"'] & & \mathsf{Trio}(\Sigma) \arrow[ddll, "1+1=1"] \\
 & \mathcal{T}_{\mathbb{Q}} & & \mathbb{B}\langle\langle\Sigma^\oplus\rangle\rangle \arrow[ll, dashed] \arrow[dl, "1+x=1"'] \arrow[dr, "a\cdot a=a"] & & & \\
\mathcal{T}_{\mathbb{N}} & & \mathsf{Sorp}(\Sigma) \arrow[ll, dashed] \arrow[dr, "a\cdot a=a"'] & & \mathsf{Why}(\Sigma) \arrow[dl, "1+x=1"'] \arrow[dr, "ab=a+b"] & & \\
 & & & \mathsf{PosBool}(\Sigma) \arrow[dr, dashed] & & \mathsf{Lin}(\Sigma) \arrow[dl, dashed] & \\
 & & & & \mathbb{B} & &
\end{tikzcd}$$

Figure 6.1: Provenance hierarchy extended with the commutative k-collapsed semiring $\mathbb{N}_k\langle\langle\Sigma^\oplus\rangle\rangle$ that can be specialized to the k-tropical semiring $\mathcal{T}_{k,\mathbb{N}}$.

Proof. Assume S is 1-bounded, then we have

$$x + xy = x \cdot (1 + y) \leqslant x \cdot (1 + 1) = x.$$

Since $x \leqslant x + xy$ trivially holds, we have $x + xy = x$. Conversely, assume that S is absorptive. Then we have

$$x + 1 = 1 + x = 1 + x \cdot 1 = 1$$

and hence, S is 1-bounded. □

This means that one can use the algorithms from [EKL08] to compute provenance over the $\mathsf{Sorp}(\Sigma)$ semiring. More specifically, it is shown in [EKL08] that over 1-bounded semirings, n fixpoint iterations are sufficient to compute the solution of n algebraic equations.

6.2.3 Computing Provenance Expressions

Computing the set-semantics of a Datalog program corresponds to computing the solution of the associated algebraic system over the Boolean semiring $(\mathbb{B}, \vee, \wedge, 0, 1)$ or equivalently, identifying all productive non-terminals of the corresponding context-free grammar.

To compute the shared expression representing the $\mathbb{N}_k\langle\langle\Sigma^\oplus\rangle\rangle$-provenance of all result facts we need to solve the algebraic system corresponding to the Datalog program over $\mathbb{N}_k\langle\langle\Sigma^\oplus\rangle\rangle$. Note that we do not need to write out the full algebraic system, but can restrict ourselves to the non-zero variables which correspond to the IDB facts under the standard set-semantics of the program. Then as a second step we compute the $\mathbb{N}_k\langle\langle\Sigma^\oplus\rangle\rangle$-provenance of all results via Newton's method applied to the algebraic system induced by the result facts.

Applying our convergence result for Newton's method over k-collapsed semirings in Proposition 3.38, we obtain the following bound on the size of the $\mathbb{N}_k\langle\langle\Sigma^\oplus\rangle\rangle$-provenance represented by shared expressions:

Theorem 6.8. *Given a Datalog program defining an IDB with n facts, the $\mathbb{N}_k\langle\langle\Sigma^\oplus\rangle\rangle$-provenance of all IDB-facts can be represented by a shared expression s of size*

$$|s| \in \mathcal{O}\left(n^3 \cdot (n + \log\log k)\right)$$

Proof. By Proposition 3.38 we need $\mathcal{O}(n + \log\log k)$ Newton iterations to compute the solution of an algebraic system comprising n equations over a k-collapsed semirings. Furthermore, every Newton iteration requires $\mathcal{O}(n^3)$ operations to solve the linearized system of n equations. □

6.2.4 Eliminating Kleene Stars from Shared Expressions

So far, we have shown how to compute shared regular expressions that compactly represent provenance information. These expressions may contain (possibly nested) Kleene stars, e.g.

$$b(ab^* + c)^*.$$

The elements of semirings like $\mathsf{Why}(\Sigma)$, $\mathsf{Lin}(\Sigma)$, or $\mathsf{Sorp}(\Sigma)$ are polynomials, and hence can be expressed without a Kleene star operation. Hence, for such semirings we may want to eliminate the Kleene star nodes from our representation to obtain "polynomial expressions" (i.e. involving only addition and multiplication nodes). Such structures have also been called "provenance circuits" recently in [DMRT14] where it was claimed that it is not possible to find polynomial sized circuits representing $\mathsf{Why}(\Sigma)$ provenance. Our main result in Theorem 6.12 refutes this claim.

Over $\mathsf{Sorp}(\Sigma)$ it is trivial to eliminate Kleene stars, since $x^* = 1$ holds by 1-boundedness. Consequently, the same is true for $\mathsf{PosBool}(\Sigma)$ which is more special than $\mathsf{Sorp}(\Sigma)$. Note that after eliminating the Kleene star over these semirings, the provenance expression only becomes smaller.

We now show that we can also eliminate the Kleene star from expressions over $\mathsf{Why}(\Sigma)$ in a space-efficient way. Since the semiring $\mathsf{Why}(\Sigma)$ is finite with $|\mathsf{Why}(\Sigma)| = 2^{|\Sigma|}$ we can trivially truncate the Kleene star of every element to a finite sum and using idempotence we obtain

$$x^* = \sum_{k=0}^{2^{|\Sigma|}} x^k = (1+x)^{2^{|\Sigma|}}.$$

Our main result in Theorem 6.10 improves this significantly and shows that the even simpler identity

$$x^* = 1 + x^{|\Sigma|}$$

holds over $\mathsf{Why}(\Sigma)$.

To this end, recall that $\mathsf{Why}(\Sigma)$ is the semiring $\mathbb{N}_\infty\langle\langle\Sigma^\oplus\rangle\rangle$ modulo idempotence of addition and $a \cdot a = a \forall a \in \Sigma$. Assume $\Sigma = \{a_1, a_2, \ldots, a_d\}$. Then modulo these additional identities a power-series

$$x = \sum_{(k_1,\ldots,k_d)\in\mathbb{N}^d} c_{(k_1,\ldots,k_d)} \cdot a_1^{k_1} \cdots a_d^{k_d} \in \mathsf{Why}(\Sigma)$$

reduces to a polynomial

$$x = \sum_{(k_1,\ldots,k_d)\in\{0,1\}^d} c_{(k_1,\ldots,k_d)} \cdot a_1^{k_1} \cdots a_d^{k_d}$$

with $c_{(k_1,\ldots,k_d)} \in \{0,1\}$. We write $\mathcal{L}(x)$ for the language of all monomials whose coefficent is 1 in x. For two power-series $x, y \in \mathsf{Why}(\Sigma)$ we write $x \leqslant y$ if $\mathcal{L}(x) \subseteq \mathcal{L}(y)$. Note that we have $x = y$ if and only if $\mathcal{L}(x) = \mathcal{L}(y)$. For two $m, m' \in \Sigma^\oplus$ with $m = a_1^{k_1} \cdots a_d^{k_d}$ and $m' = a_1^{k'_1} \cdots a_d^{k'_d}$ we write $m \leqslant m'$ if $(k_1, \ldots, k_d) \leqslant (k'_1, \ldots, k'_d)$.

As mentioned before, the $\mathsf{Why}(\Sigma)$ semiring is also isomorphic to

$$(2^{2^\Sigma}, \cup, \uplus, \emptyset, \{\emptyset\})$$

with $X \uplus Y = \{A \cup B : A \in X, B \in Y\}$ for $X, Y \in 2^{2^\Sigma}$.

For $x \in 2^{2^\Sigma}$, let $N(x)$ denote the number of different alphabet symbols appearing in x, i.e. formally

$$N(x) := \left| \bigcup_{x_i \in x} x_i \right|.$$

The following lemma contributes the main ingredient for our Theorem:

Lemma 6.9. *Let* $x, y \in \mathsf{Why}(\Sigma)$ *with* $N = N(xy)$. *Then*

$$(xy)^* \leqslant 1 + (x+y)^N$$

Proof. We first show

$$(xy)^* = \sum_{k\in\mathbb{N}} (xy)^k \leqslant 1 + (xy)^N$$

Because of idempotence it suffices to show that for any $k \in \mathbb{N}$ we have $(xy)^k \leqslant 1+(xy)^N$. If $k = 0$, we obviously have $(xy)^0 = 1 \leqslant 1 + (xy)^N$. So, assume $k > 0$ in the following and pick any $m \in \mathcal{L}((xy)^k)$. Then there exist $u_1, \ldots, u_k \in \mathcal{L}(x)$ and $v_1, \ldots, v_k \in \mathcal{L}(y)$ such that

$$m = u_1 v_1 \cdots u_k v_k = u_1 \ldots u_k v_1 \cdots v_k$$

If $k < N$, we have

$$m = u_1 \cdots u_k^{N-k+1} v_1 \cdots v_k^{N-k+1} \in \mathcal{L}((xy)^N).$$

as $x^2 = x$ for all $x \in \Sigma^*$. If $k > N$, we note that the longest strictly $\leqslant$-increasing sequence of monomials

$$m_0 < m_1 < m_2 < \ldots < m_l$$

can consist of at most $N + 1$ elements: starting from the empty word, m_i and m_{i+1} have to differ in at least one symbol. Hence, the products

$$u_1 \ldots u_k \quad \text{and} \quad v_1 \cdots v_k$$

can be reduced to products consisting of at most N non-zero monomials, i.e. we again have $\mathcal{L}((xy)^k) \subseteq \mathcal{L}((xy)^N)$. For the same reason, the combined product

$$u_1 \ldots u_k \cdots v_1 \cdots v_k$$

can also be reduced to a product consisting of at most N non-zero monomials. From this observation the second inequality follows. □

Using the previous lemma, we can inductively eliminate Kleene stars over $\mathsf{Why}(\Sigma)$:

Theorem 6.10. *For any $x \in \mathsf{Why}(\Sigma)$ containing $N = N(x)$ different alphabet symbols we have*

$$x^* = 1 + x^N.$$

Proof. The inequality $1 + x^{N(x)} \leqslant x^*$ is immediate. We prove the converse inequality by structural induction on x:

- Case $x = a \in \Sigma$ or $x = 1$: trivial.
- Case $x = u + v$: $x^* = u^* v^* \overset{IH}{=} (1 + u^N)(1 + v^N) = 1 + u^N + v^N + u^N v^N \leqslant 1 + (u + v)^N = 1 + x^N$
- Case $x = u \cdot v$: shown by Lemma 6.9.
- Case $x = u^*$: $(u^*)^* = u^* = 1 + u^N \leqslant 1 + x^N$ by the inductive hypothesis.

□

Corollary 6.11. *The semiring $\mathsf{Why}(\Sigma)$ is $|\Sigma|$-closed, i.e. $x^* = \sum_{i=0}^{|\Sigma|} x^i$ for all $x \in \mathsf{Why}(X)$.*

Using Theorem 6.10 we can eliminate any Kleene star from an expression x by replacing it with $N = N(x)$ multiplications. Using the idea of repeated squaring for fast exponentiation we can encode any expression x^N by introducing $\mathcal{O}(\log(N))$ new multiplication nodes. Hence, every Kleene star appearing in the expression gives rise to at most $\mathcal{O}(\log(N))$ additional nodes. The following theorem sums up our discussion:

Theorem 6.12. *Every shared expression* r *encoding provenance over* $\mathsf{Why}(\Sigma)$ *can be transformed into an equivalent expression* r' *without Kleene star nodes having size* $|r'| \in \mathcal{O}(|r| \cdot \log|\Sigma|)$.

Example 6.13. *Consider the following expression with nested Kleene stars*

$$r = (ab^* + c)^*.$$

Over $\mathsf{Why}(\Sigma)$ *this expression is equivalent to*

$$r = (1 + a(1 + b) + c)^3.$$

The following pictures show the representation of the term r *as shared expressions with Kleene star nodes (left) and the result after eliminating the stars (right). The topmost (boxed) node in each structure corresponds to the expression* r.

$$\begin{tikzcd}
 & & & & & & & \boxed{\cdot} \arrow[dl] \arrow[ddl, bend left] \\
 & & & & & & \cdot \arrow[d, bend right] \arrow[d, bend left] & \\
 & \boxed{*} \arrow[d] & {} \arrow[r, "\text{elim. stars}"] & {} & & & + \arrow[ddddll, bend right] \arrow[dr] & \\
 & + \arrow[d] \arrow[dddr] & & & & & & + \arrow[dl] \arrow[ddd] \\
 & \cdot \arrow[dl] \arrow[dd] & & & & & \cdot \arrow[dl] \arrow[dd] & \\
 * \arrow[d] & & & & & + \arrow[dl] \arrow[d] & & \\
 b & a & c & & 1 & b & a & c
\end{tikzcd}$$

6.3 Related Work and Conclusions

The philosophy of solving different analysis problems over graphs in a uniform way by first computing a general abstract expression and then obtaining the specific results by applying a homomorphism to this expression goes back (at least) to Tarjan [Tar81]. He considers "path problems" over (finite) directed graphs (shortest paths, solving linear equations, dataflow analysis problems) and proposes to represent the set of all paths via regular expressions which can then be evaluated by problem specific homomorphisms.

In [STL13] we described the concise shared expression data structure that is used by FPSOLVE to represent terms involving the semiring operations of addition, multiplication, and Kleene star. Deutch et al. [DMRT14] also propose such shared structures (termed "circuits") to represent provenance information. However, since their circuits only comprise addition and multiplication nodes they cannot finitely represent elements from the more general semiring $\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle$.

In this chapter we have shown that our theoretical results on Newton's method can be fruitfully applied to provenance analyses for recursive Datalog programs. In particular we make the following contributions:

1. Extending the work of [GKT07] we propose to separate the issues of specifying and computing provenance: Provenance is *specified* by an algebraic system over a semiring that arises directly from the Datalog program. For *computing* provenance we can choose between different algorithms (e.g. fixpoint iteration or Newton's method).

2. We identify the semiring $\mathbb{N}_k\langle\langle\Sigma^{\oplus}\rangle\rangle$ as a very general semiring that can be specialized to various provenance semirings used in applications.

3. We show how to represent provenance information over $\mathbb{N}_k\langle\langle\Sigma^{\oplus}\rangle\rangle$ concisely via shared expressions and how to compute them via Newton's method.

4. Our main result is that over semirings such as $\mathsf{Why}(\Sigma)$ we can specialize the shared expressions to eliminate Kleene star nodes without sacrificing the succinctness of the representation.

Note that these results have (briefly) appeared in [LS13, LS14, LS15].

7 Application II: Subword and Superword Closure of Context-Free Languages

In this chapter we study the subword (and superword) closure of context-free languages, especially the problem of succinctly representing these closures. The connection to algebraic systems is detailed in [EKL07a]: The subword closure of a CFL is the solution of an algebraic system over a "lossy" semiring (a semiring where $x = 1 + x$ holds). Since algebraic systems and grammars are two sides of the same coin we mostly assume the view of formal language theory.

First, we review Courcelle's construction and present it in a more structured way as grammar transformation of CFGs into *simple C2F* in Section 7.2. We then study three different *representations* of the subword closure of CFGs in Section 7.3: NFAs, DFAs, and regular expressions. We present asymptotically tight bounds on the descriptional complexity for all three formalisms. In Section 7.4 we present an application to CFG equivalence testing and prove that NFA equivalence modulo subword closure is (only) NP-complete. This is a surprising result, since NFA equivalence is PSPACE-complete even for languages that are prefix, suffix, or factor-closed [KRS09, RSX12].

All results of this chapter have appeared in [BLS15], except for the extension to regular expressions in Section 7.3.3 which constitutes original work.

7.1 Introduction

We call a word $w = a_1 \cdots a_k \in \Sigma^k$ a *subword* (also: *scattered subword* or *subsequence*) of $u \in \Sigma^*$ if we can write u as $u = u_0 a_1 u_1 a_2 \cdots u_{k-1} a_k u_k$ with $u_i \in \Sigma^*$. Accordingly, we say that u is a *superword* of w. We write $w \preccurlyeq u$ if w is a subword of u.

For a language $L \subseteq \Sigma^*$ we define its *subword closure* ∇L as the set of all subwords of words in L, i.e. $\nabla L := \{w \in \Sigma^* : \exists u \in L\; w \preccurlyeq u\}$. Analogously, the *superword closure* ΔL is defined as $\Delta L := \{u \in \Sigma^* : \exists w \in L\; w \preccurlyeq u\}$. For a single word $w = a_1 \cdots a_k \in \Sigma^k$ we define its superword closure as the language $\Delta w := \Sigma^* a_1 \Sigma^* \cdots \Sigma^* a_k \Sigma^*$.

It is a well-known fact (Higman's lemma) that the subword order $\preccurlyeq$ is a well-quasi order, i.e. every set of elements has only finitely many minimal elements [Hig52]. Hence, the superword closure of *any* language L has a finite set of minimal elements $w_1, \ldots, w_n$ and we can represent ΔL as $\bigcup_{i \in [n]} \Delta w_i$. In particular, the superword closure of *any* language is a regular language. This might seem surprising, however since $L = \emptyset \Leftrightarrow \Delta L = \emptyset$ the superword closure is in general not computable given a description of a language (e.g. as a grammar): Consider for instance the class of context-sensitive languages for which emptiness is undecidable.

Note that the complement of a subword-closed language is *a* superword-closed language (but in general $\Sigma^* \setminus \nabla L \neq \Delta L \setminus L$) and hence, also the subword-closure ∇L is regular for any language L. As for the superword closure, it is not possible in general to compute a finite automaton representing the subword closure of an arbitrary language. However, for special classes of languages like the context-free languages or Petri net languages there are effective procedures for computing the subword closure [vL78, Cou91, HMW10]. More recently, Zetsche has discovered a very general method to compute subword closures, in particular it is shown in his seminal paper [Zet15] that the subword closure of indexed languages [Aho68] is effectively computable.

7.2 An Optimized Variant of Courcelle's Construction

We describe a variation of Courcelle's construction for computing the subword closure of a context-free languages [Cou91]. Given a CFG $G = (\mathcal{X}, \Sigma, \rightarrow)$ and a nonterminal $X \in \mathcal{X}$, we simply write ∇X for $\nabla \mathcal{L}(X)$. Note that in this chapter we assume that every grammar has a designated *start symbol* S such that we can define $\mathcal{L}(G) := \mathcal{L}(S)$. We write the grammar as $G = (\mathcal{X}, \Sigma, \rightarrow, S)$ if we need to refer to the start symbol explicitly. We assume w.l.o.g. that G does not contain any unproductive nonterminals, i.e. $\mathcal{L}(X) \neq \emptyset$ for all $X \in \mathcal{X}$. By $\Sigma_X \subseteq \Sigma$ we denote the portion of the alphabet that is *reachable* from X, i.e. $a \in \Sigma_X$ if and only if $X \Rightarrow_G^* uav$ for some $u, v \in \Sigma^*$.

The *dependency graph* associated with a CFG G is the finite directed graph with nodes $\mathcal{X}$ and an edge from X to Y if there is a production of the form $X \rightarrow_G \alpha Y \beta$ in G We write $X \rhd Y$ ("X depends directly on Y") if $X \neq Y$ and there is an edge from X to Y in the dependency graph. If X and Y are located in the same strongly connected component of the dependency graph we write $X \equiv Y$, i.e $X \equiv Y$ if and only if $X \trianglerighteq^* Y \wedge Y \trianglerighteq^* X$.

Courcelle's construction rests on the following properties of the subword closure:

Lemma 7.1 ([Cou91]). *Given a context-free grammar $G = (\mathcal{X}, \Sigma, \rightarrow)$ and nonterminals $X, Y \in \mathcal{X}$, the following hold:*

1. *If $X \Rightarrow_G^* \alpha X \beta X \gamma$ for $\alpha, \beta, \gamma \in (\mathcal{X} \cup \Sigma)^*$ then $\nabla X = \Sigma_X^*$.*
2. *If $X \equiv Y$ then $\nabla X = \nabla Y$.*
3. *The ∇-operation distributes over union and concatenation: $\nabla(\mathcal{L}(X) \cup \mathcal{L}(Y)) = \nabla X \cup \nabla Y$ and $\nabla(\mathcal{L}(X) \cdot \mathcal{L}(Y)) = \nabla X \cdot \nabla Y$.*

Example 7.2 ([BLS15]). *Consider the CFG in Figure 7.1 over the alphabet $\Sigma = \{a, b, c\}$. First,*

$$
\begin{array}{llll}
S & \rightarrow XaU \mid VaU \mid X & X & \rightarrow ZbY \mid \varepsilon \\
Y & \rightarrow XYa \mid b & U & \rightarrow VZ \mid acb \\
V & \rightarrow ZU \mid \varepsilon & Z & \rightarrow cZ \mid bc
\end{array}
$$

$$\begin{tikzcd}
& & & S \arrow[dll] \arrow[d] & \\
Y \arrow[loop left] \arrow[r, bend left] & X \arrow[l, bend left] \arrow[drr] & & U \arrow[r, bend left] \arrow[d] & V \arrow[l, bend left] \arrow[dl] \\
& & & Z \arrow[loop right] &
\end{tikzcd}$$

Figure 7.1: Example grammar with dependency graph.

note that Y can derive two copies of itself via $Y \Rightarrow XYa \Rightarrow ZbYYa$ so its subword closure is given by $\nabla Y = \Sigma_Y^*$, where $\Sigma_Y = \{a, b, c\}$. Considering that X and Y are mutually reachable we get $\nabla X = \nabla Y$. Next, since the ∇-operation distributes over union and concatenation we obtain from the language-defining equation $Z = \{c\}Z \cup \{bc\}$ that ∇Z is the solution to

$$
\begin{aligned}
&\nabla Z = \nabla(c)\nabla Z + \nabla(bc) = (c + \varepsilon)\nabla Z + (bc + b + c + \varepsilon) \\
\Rightarrow\ &\nabla Z = c^*(bc + b + c + \varepsilon)
\end{aligned}
$$

As U and V reside in the same SCC of the dependency graph we have $\nabla U = \nabla V$ and hence ∇U is the solution of the equation

$$
\begin{aligned}
&\nabla U = \nabla U \nabla Z + \nabla Z \nabla U + \nabla(acb) + \varepsilon \\
\Rightarrow\ &\nabla U = \Sigma_Z^* \nabla(acb) \Sigma_Z^* \\
&\quad\ = c^*(acb + ac + ab + cb + a + b + c + \varepsilon)c^*
\end{aligned}
$$

From these results we can finally obtain ∇S (for which the expression is so large that we refrain from expanding it):

$$\begin{aligned}\nabla Z &= c^*(bc + b + c + \varepsilon)\\ \nabla X = \nabla Y &= (a + b + c)^*\\ \nabla U = \nabla V &= c^*(acb + ac + ab + cb + a + b + c + \varepsilon)c^*\\ \nabla S &= \nabla X(a + \varepsilon)\nabla U + \nabla V(a + \varepsilon)\nabla U + \nabla X.\end{aligned}$$

As one can see from this example, the descriptions of the subword closure can grow quite large and it is not immediately obvious how to derive good upper bounds for them. Next, we present a special normal form for CFGs that allows us to derive optimal bounds on the sizes of different representations (NFAs, DFAs, regular expressions) for the subword closure of a CFG in a systematic way.

7.2.1 Preprocessing the Grammar

As noted in Section 2.3.2 we can transform every grammar into canonical two form (C2F) with at most a linear blowup in size and without changing its language. The next step is to transform the grammar into *simple C2F*, which changes G's language but not its subword closure.

Definition 7.3 (Simple C2F). *We call a grammar in C2F* simple *if it satisfies the following:*

- *For each $x \in \Sigma \cup \{\varepsilon\}$ there is a unique nonterminal A_x with the only production $A_x \to x$.*
- *Every other production is in one of the following forms:*
  - $X \to XY$ *with* $Y \not\equiv X$*, or*
  - $X \to Y$ *with* $Y \not\equiv X$*, or*
  - $X \to YZ$ *with* $Y \not\equiv X \wedge Z \not\equiv X$.

First, we need a small lemma that allows us to eliminate all linear rules "within" some SCC, i.e. rules of the form $X \to \alpha Y \beta$ such that $X \neq Y$ but $Y \trianglerighteq^* X$. We call a grammar *strongly connected* if its dependency graph is strongly connected.

Lemma 7.4. *Let* G *be a strongly connected linear CFG with nonterminals $\mathcal{X} = \{X_1, \ldots, X_n\}$ so that every rule is either of the form $X \to \alpha Y \beta$ or $X \to \alpha$ for $\alpha, \beta \in \Sigma^*$. Consider the grammar G' which we obtain from* G *by replacing in every rule of* G *every occurrence of a nonterminal X_i by* Z. *We then have that $\nabla\mathcal{L}(Z) = \nabla\mathcal{L}(X_i)$ for all $i \in [n]$.*

Proof. Since G is strongly connected, $\nabla X_i = \nabla X_j$ for all $i, j \in [n]$, hence it suffices to show the statement for X_1. Clearly, $\mathcal{L}(Z) \supseteq \mathcal{L}(X_1)$ hence also $\nabla Z \supseteq \nabla X_1$. For the other inclusion let $w \in \nabla Z$, i.e. we have a word w' with $w \preccurlyeq w' \in \mathcal{L}(Z)$ possessing some derivation $Z \Rightarrow u_0 Z v_0 \Rightarrow u_0 u_1 Z v_1 v_0 \Rightarrow \cdots \Rightarrow w'$. Since G is strongly connected there must be an X_{j_1} reachable from X_1 with $X_{j_1} \to u_0 X_{k_1} v_0$ for some Y. Continuing this reasoning we generate a superword of w' (with some "junk"-strings α_l, β_l) by following the derivation of w':

$$X_1 \Rightarrow^* \alpha_0 X_{j_1} \beta_0 \Rightarrow \alpha_0 u_0 X_{k_1} v_0 \beta_0 \Rightarrow^* \alpha_0 u_0 \alpha_1 u_1 X_{k_2} v_1 \beta_1 v_0 \beta_0 \Rightarrow \cdots \Rightarrow w''$$

with $w' \preccurlyeq w''$. Since, $w \preccurlyeq w'$ we have $w \in \nabla X_1$. □

Lemma 7.5. *For every CFG* G *there is a CFG* G′ *in simple C2F such that* $|G'| \in \mathcal{O}(|G|)$ *and* $\nabla\mathcal{L}(G) = \nabla\mathcal{L}(G')$*.*

Proof. The following steps achieve the desired result:

1. For every $x \in \Sigma \cup \{\varepsilon\}$ replace every occurrence of x in a rule by A_x and finally add the rule $A_x \to x$.

2. For every rule $X \to \alpha Y \beta Z \gamma$ with $Y, Z \equiv X$ replace all rules with left-hand-side Y such that $Y \equiv X$ (i.e. from the same SCC as X) by the rules $X \to A_x X$ for all $x \in \Sigma_X$ and add $X \to A_\varepsilon$.

3. Binarize the grammar by introducing fresh nonterminals, such that every rule is of the form $X \to \alpha$ with $|\alpha| \leqslant 2$.

4. Contract every strongly connected component of the grammar into a univariate subgrammar via Lemma 7.4 [1].

It is easy to check that G′ is indeed in simple C2F, moreover steps (1) and (3) do not change the language of the grammar. In step (2) we ensure that $\mathcal{L}(X) = \Sigma_X^*$ if $X \Rightarrow^* \alpha X \beta X \gamma$ (see Lemma 7.1). Step (4) also preserves the subword closure (by Lemma 7.4), thus altogether $\nabla\mathcal{L}(G) = \nabla\mathcal{L}(G')$. Step (2) reduces the size of G, steps (1) and (3) lead to a linear growth, and step (4) does not change the size so together there exists a constant c (independent of G) such that $|G'| \leqslant c \cdot |G|$. □

Example 7.6. *Consider again the grammar from the previous example*

$$\begin{array}{lll} S & \to XaU \mid VaU \mid X & \quad X \to ZbY \mid \varepsilon \\ Y & \to XYa \mid b & \quad U \to VZ \mid acb \\ V & \to ZU \mid \varepsilon & \quad Z \to cZ \mid bc \end{array}$$

$$\begin{tikzcd}
 & & S \arrow[dl] \arrow[d] & \\
Y \arrow[loop left] \arrow[r, bend left] & X \arrow[l, bend left] \arrow[dr] & U \arrow[r, bend left] \arrow[d] & V \arrow[l, bend left] \arrow[dl] \\
 & & Z \arrow[loop right] &
\end{tikzcd}$$

[1] Here we implicitly treat nonterminals from lower SCCs as terminals, since CFLs are closed under substitution this is fine.

As in the previous example, we use the fact $\nabla X = \nabla Y = \{a, b, c\}^*$ *to eliminate* Y *from the grammar and to change the rules for* X *to* $X \to aX \mid bX \mid cX \mid \varepsilon$ *– this clearly preserves* ∇X*. Then since* U *and* V *are in the same SCC, we can rename* V *to* U *and collect all their defining rules. Finally, we introduce fresh nonterminals* W *(for* aU*) and* T *(for* cb*) to binarize the grammar and replace every terminal* x *by a fresh nonterminal* A_x*. The transformed grammar and its dependency graph now look like this:*

$$\begin{array}{llll}
S & \to XW \mid UW \mid X & W & \to A_aU \\
X & \to A_aX \mid A_bX \mid A_cX \mid A_\varepsilon & U & \to UZ \mid ZU \mid A_aT \mid A_\varepsilon \\
T & \to A_cA_b & Z & \to A_cZ \mid A_bA_c \\
A_a & \to a & A_b & \to b \\
A_c & \to c & A_\varepsilon & \to \varepsilon
\end{array}$$

$$\begin{tikzcd}
 & S \arrow[dl] \arrow[d] \arrow[dr] & & \\
X \arrow[loop left] \arrow[ddd] \arrow[dddr] \arrow[dddrr] \arrow[dddrrr] & W \arrow[r] \arrow[dd] & U \arrow[loop right] \arrow[d] \arrow[dr] \arrow[ddl] \arrow[ddll] & \\
 & & T \arrow[d] \arrow[dr] & Z \arrow[loop right] \arrow[d] \arrow[dl] \\
A_\varepsilon & A_a & A_c & A_b
\end{tikzcd}$$

Note that the dependency graph is a directed acyclic graph (DAG) apart from the self-loops.

7.3 Representations for the Subword Closure of CFLs

Since the subword closure of a CFL is a regular language, any standard representations for regular languages such as finite automata or regular expressions can be used to represent it. In [GHK09] Gruber, Holzer, and Kutrib have studied the size of finite automata representations for the subword closure of CFLs and proved an upper bound on the size of the NFA-representation of $2^{2^{\mathcal{O}(|G|)}}$ (together with a well-known lower bound of $2^{\Omega(|G|)}$). In the same article they posed the open problems (1) to tighten this bound further, and (2) to study the size of the (minimal) DFA representation of the subword closure. Our results give answers to both questions and provide asymptotically tight bounds on the size of finite automata representations for the subword closure of CFLs, thereby completing the study in [GHK09]. We further show that our results can be extended to regular expressions using a slightly different construction (cf. Section 7.3.3).

In [ABT08] it is remarked that a straightforward implementation of Courcelle's construction yields an NFA of "single exponential" size w.r.t. $|G|$. However, no detailed complexity analysis is given. Unfortunately, a straightforward implementation of Courcelle's construction yields a suboptimal bound of $2^{\Omega(|G| \log |G|)}$ for the size of an NFA recognizing ∇G. To illustrate this, consider the CFG of size $\mathcal{O}(n)$ with start-symbol A_n

comprising the rules

$$\begin{aligned}
&A_0 \to a\\
&A_1 \to A_0A_0\\
&A_2 \to A_0A_0 \mid A_0A_1 \mid A_1A_0 \mid A_1A_1\\
&\quad\vdots\\
&A_n \to A_iA_j \quad \forall 0 \leqslant i,j \leqslant (n-1).
\end{aligned}$$

If we compute an NFA for ∇A_n via the straight-forward bottom-up construction it is of size $a_n := |\mathcal{A}_{\nabla A_n}|$ with $a_0 = 2$ and

$$a_n = 2 + \sum_{0 \leqslant i,j \leqslant (n-1)} (a_i + a_j) = 2 + 2n \sum_{i=0}^{n-1} a_i.$$

Setting $s_n := \sum_{i=0}^{n} a_i$ we obtain

$$\begin{aligned}
s_n - s_{n-1} &= a_n = 2 + 2ns_{n-1}\\
&\Rightarrow s_n = 2 + (2n+1)s_{n-1}.
\end{aligned}$$

As $s_i \geqslant 2$ we can bound s_n by $2ns_{n-1} \leqslant s_n \leqslant 2(n+1)s_{n-1}$ and by induction on n we get

$$2^{n+1}n! \leqslant s_n \leqslant 2^{n+1}(n+1)!$$

From this we obtain the lower bound on $a_n = |\mathcal{A}_{\nabla A_n}|$:

$$a_n = s_n - s_{n-1} \geqslant 2^{n+1}n! - 2^n n! = 2^n n! \in 2^{\Omega(n \log n)}.$$

Hence, the crucial part to achieve the optimal bound of $2^{\mathcal{O}(|G|)}$ for NFAs is to reuse already computed automata.

7.3.1 NFAs

Here we describe a space efficient version of Courcelle's construction that exploits the simple C2F and re-uses already computed automata in order to achieve an asymptotically tight bound on the size of the NFA.

Theorem 7.7. *For any CFG G in simple C2F with n nonterminals there is an NFA $\mathcal{A}$ with at most $2 \cdot 3^{n-1}$ states which recognizes the subword closure of G, i.e. $\nabla\mathcal{L}(G) = \mathcal{L}(\mathcal{A})$.*

Before we describe the proof, we collect some definitions used:

Definition 7.8. *Given a nonterminal X in a grammar $G = (\mathcal{X}, \Sigma, \to_G)$ in simple C2F, we define the following sets of nonterminals and terminals:*

- $Q(X) := \{YZ \in \mathcal{X} \cdot \mathcal{X} : X \to_G YZ\}$ *("quadratic monomials")*
- $L(X) := \{Y \in \mathcal{X} : X \to_G Y\}$ *("linear monomials")*
- $C_l(X) := \{Y \in \mathcal{X} : X \to_G YX\}$ *("left coefficients")*
- $C_r(X) := \{Y \in \mathcal{X} : X \to_G XY\}$ *("right coefficients")*
- $\Sigma_l(X) := \Sigma \cap \bigcup\{\nabla\mathcal{L}(Y) : Y \in C_l(X)\}$ *("left alphabet")*
- $\Sigma_r(X) := \Sigma \cap \bigcup\{\nabla\mathcal{L}(Y) : Y \in C_r(X)\}$ *("right alphabet")*

Note that $\Sigma_l(X)$ (resp. $\Sigma_r(X)$) is simply the set of terminals reachable from any element of $C_l(X)$ (resp. $C_r(X)$), and can therefore be easily computed.

Proof. We prove the statement by induction on n. Each NFA representing the subword closure of a nonterminal has two designated states, called the entry and exit state, respectively. Furthermore, all states are final. For $n = 1$ the grammar comprises only one nonterminal A_a and the rule $A_a \to a$, its subword closure is therefore $\{\varepsilon, a\}$ which can be recognized by the following 2-state NFA:

$$\begin{tikzcd}
\text{start} \arrow[r] & q_{en} \arrow[r, "{\varepsilon, a}"] & q_{ex}
\end{tikzcd}$$

Let now $n > 1$. Consider the start symbol S of the grammar. The language of the grammar is the least solution of

$$\mathcal{L}(S) = \bigcup_{X \in C_l(S)} \mathcal{L}(X) \cdot \mathcal{L}(S) \cup \bigcup_{XY \in Q(S)} \mathcal{L}(X) \cdot \mathcal{L}(Y) \cup \bigcup_{Y \in L(S)} \mathcal{L}(Y) \cup \bigcup_{Y \in C_r(S)} \mathcal{L}(S) \cdot \mathcal{L}(Y).$$

Using Lemma 7.1, the subword closure of S is obtained as

$$\nabla S = \bigcup_{X \in C_l(S)} \nabla X \cdot \nabla S \cup \bigcup_{XY \in Q(S)} \nabla X \cdot \nabla Y \cup \bigcup_{Y \in L(S)} \nabla Y \cup \bigcup_{Y \in C_r(S)} \nabla S \cdot \nabla Y.$$

The solution of this equation is

$$\nabla S = \Sigma_l(S)^* \cdot \left(\bigcup_{XY \in Q(S)} \nabla X \cdot \nabla Y \cup \bigcup_{Y \in L(S)} \nabla Y \right) \Sigma_r(S)^*.$$

As the dependency graph of a grammar in simple C2F is acyclic apart from self-loops, we can topologically order the set $\mathcal{X}$ of nonterminals w.r.t. the dependency graph. Let the nonterminals be indexed with their position in the reverse ordering, i.e. $X_1, \ldots, X_n$ such that $S = X_n$. Then the number of nonterminals reachable from each X_i is bounded by i.

Since the grammar is in simple C2F, we know that S is not reachable from any $X_i \in Q(S) \cup L(S)$ and therefore, the subgrammar of G induced by any $X_i \in Q(S) \cup L(S)$

contains at most $i < n$ nonterminals[2]. Hence by induction we can compute for each $X_i \in Q(S) \cup L(S)$ an NFA recognizing ∇X_i of size at most $2 \cdot 3^{i-1}$.

Now the construction of the NFA $\mathcal{A}_S$ for ∇S works as follows:

- For each X in the set $\{X : XY \in Q(S)\}$ compute the NFA for ∇X and call it $\mathcal{A}_X^{(1)}$.
- For each Y in the set $\{Y : XY \in Q(S) \vee Y \in L(S)\}$ compute the NFA for ∇Y and call it $\mathcal{A}_Y^{(2)}$.

Initially we let $\mathcal{A}_S$ be the disjoint union of all $\mathcal{A}_X^{(1)}, \mathcal{A}_X^{(2)}$ (suitably renaming all entry/exit states of the automata). Furthermore, we create two new final states q_{en}, q_{ex} and set q_{en} as the initial state of $\mathcal{A}_S$. Finally, we add the following transitions to the automaton:

- For each $XY \in Q(S)$: Add ε-transitions (1) from q_{en} to the entry state of $\mathcal{A}_X^{(1)}$, (2) from the exit state of $\mathcal{A}_X^{(1)}$ to the entry state of $\mathcal{A}_Y^{(1)}$, and (3) from the exit state of $\mathcal{A}_Y^{(2)}$ to q_{ex}.
- For each $X \in L(S)$: Add ε-transitions (1) from q_{en} to the entry state of $\mathcal{A}_X^{(2)}$, and (2) from the exit state of $\mathcal{A}_X^{(2)}$ to q_{ex}.
- For each $a \in \Sigma_l(S)$ add a self-loop $q_{en} \xrightarrow{a} q_{en}$.
- For each $a \in \Sigma_r(S)$ add a self-loop $q_{ex} \xrightarrow{a} q_{ex}$.

Note that in our construction we create at most two copies $\mathcal{A}_X^{(1)}, \mathcal{A}_X^{(2)}$ of each NFA for $X \in Q(S) \cup L(S)$. Since $S = X_n$ we have $Q(S) \cup L(S) \subseteq \{X_1, \ldots, X_{n-1}\}$ and by induction $|\mathcal{A}_{X_i}| \leqslant 2 \cdot 3^{i-1}$. Hence the size of $\mathcal{A}_S$ is bounded by

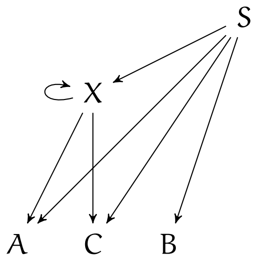
$$\begin{aligned}
|\mathcal{A}_S| &\leqslant 2 + \sum_{X \in Q(S) \cup L(S)} 2 \cdot |\mathcal{A}_X| \\
&\leqslant 2 + \sum_{X \in \{X_1, \ldots, X_{n-1}\}} 2 \cdot |\mathcal{A}_X| \\
&\leqslant 2 + \sum_{i=1}^{n-1} 2 \cdot \left(2 \cdot 3^{i-1}\right) \\
&= 2 + 4 \cdot \sum_{i=0}^{n-2} 3^i \\
&= 2 + 4 \left(\cdot \frac{3^{n-1} - 1}{2}\right) \\
&= 2 \cdot 3^{n-1}
\end{aligned}$$

□

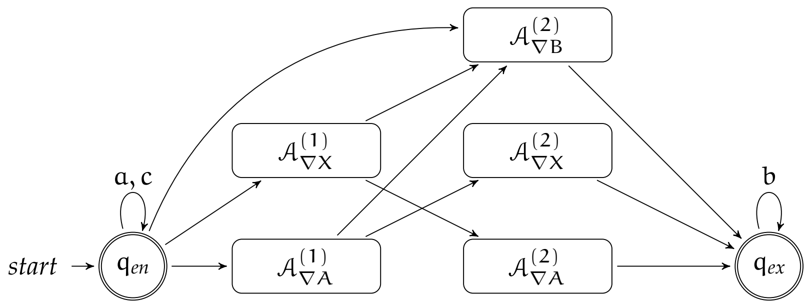
[2]By "subgrammar induced by X" we mean the subgrammar of G that contains all nonterminals that reachable from X in the dependency graph of G.

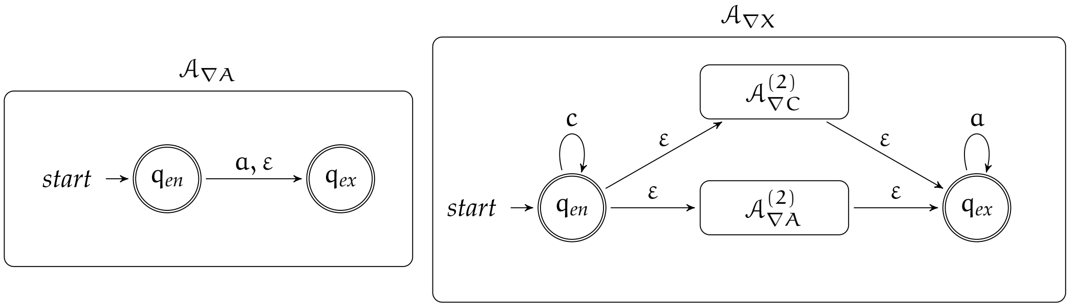
Example 7.9. *Consider the following grammar in simple C2F together with its dependency graph.*

$$
\begin{aligned}
S &\to XS \mid SB \mid XA \mid AB \mid AX \mid XB \mid B\\
X &\to CX \mid XA \mid C \mid A\\
A &\to a\\
B &\to b\\
C &\to c
\end{aligned}
$$



The automaton for ∇S *is depicted in the top, the automaton for* ∇A *in the lower left, and the automaton for* ∇X *in the lower right picture. We obtain the NFA for* ∇S *by "wiring" together (with* ε*-edges) the corresponding automata representing the "lower" nonterminals. For example, the rule* $S \to AX$ *gives rise to the connections* $q_{en} \xrightarrow{\varepsilon} \mathcal{A}^{(1)}_{\nabla A} \xrightarrow{\varepsilon} \mathcal{A}^{(2)}_{\nabla X} \xrightarrow{\varepsilon} q_{ex}$*. Moreover, the rule* $S \to XS$ *leads to a self-loop labeled with* $\Sigma_X = \{a, c\}$ *at the entry state* q_{en} *of* $\mathcal{A}_{\nabla S}$*.*


$\mathcal{A}^{(2)}_{\nabla B}$
$\mathcal{A}^{(1)}_{\nabla X}$
$\mathcal{A}^{(2)}_{\nabla X}$
a, c
b
start
q_{en}
$\mathcal{A}^{(1)}_{\nabla A}$
$\mathcal{A}^{(2)}_{\nabla A}$
q_{ex}



7.3.2 DFAs

As the size of an NFA recognizing $\nabla\mathcal{L}(G)$ is bounded by $2^{\mathcal{O}(|G|)}$ we immediately obtain an upper bound of $2^{2^{\mathcal{O}(|G|)}}$ for the size of the minimal DFA recognizing $\nabla\mathcal{L}(G)$. Next we show that this bound is essentially tight. To this end consider the following family of finite languages L_k of words $w \in \{0,1\}^{2k+1}$ such that $w = x0y0z$ for $y \in \{0,1\}^k, x, z \in \{0,1\}^*$. These are all words of length $2k+1$ which contain two zeros that are exactly k letters apart. We can write L_k as

$$L_k = \bigcup_{j=1}^{k} \{0,1\}^{j-1} \cdot \{0\} \cdot \{0,1\}^{k} \cdot \{0\} \cdot \{0,1\}^{k-j}.$$

We are particularly interested in L_k for $k = 2^n$. The following CFG G_n of size $\mathcal{O}(n)$ with start symbol X'_n provides a succinct representation of L_{2^n}, i.e. $L(X'_n) = L_{2^n}$:

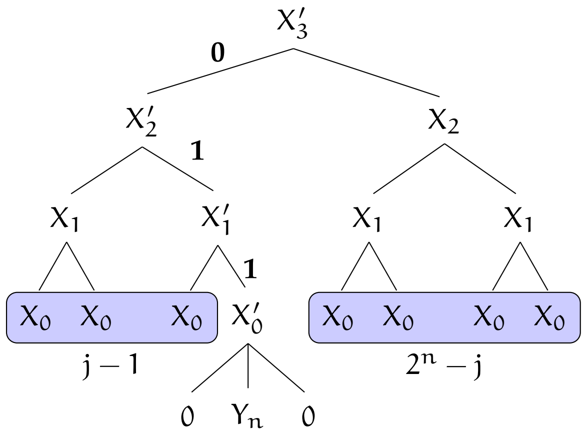
$$\begin{array}{lcl@{\qquad\qquad}lcl}
X'_n & \to & X'_{n-1}X_{n-1} \mid X_{n-1}X'_{n-1} & & & \\
X'_{n-1} & \to & X'_{n-2}X_{n-2} \mid X_{n-2}X'_{n-2} & X_{n-1} & \to & X_{n-2}X_{n-2} \\
 & \vdots & & & \vdots & \\
X'_1 & \to & X'_0X_0 \mid X_0X'_0 & X_1 & \to & X_0X_0 \\
X'_0 & \to & 0Y_n0 & X_0 & \to & 0 \mid 1 \\
Y_n & \to & Y_{n-1}Y_{n-1} & & & \\
Y_{n-1} & \to & Y_{n-2}Y_{n-2} & & & \\
 & \vdots & & & & \\
Y_1 & \to & Y_0Y_0 & & & \\
Y_0 & \to & 0 \mid 1 & & &
\end{array}$$

The grammar uses repeated squaring to achieve the required compression while a derivation uses the "primed" nonterminals X'_i to choose where to insert a word from the set $\{0\}\{0,1\}^{2^n}\{0\}$ into a word from $\{0,1\}^{2^n}$. The two alternative rules for a primed nonterminal allow the derivation to choose whether the left or the right child becomes the new primed nonterminal.

Example 7.10. *We want to illustrate that each word from $\{0,1\}^{j-1}\{0\}\{0,1\}^{2^n}\{0\}\{0,1\}^{2^n-j}$ can be derived by the above grammar. The idea is that the n-bit binary expansion of $j-1$ encodes the path to the node labeled X'_0 in a derivation tree of the grammar.*

The picture shows a partial derivation tree for $n = 3$ and $j = 4$. Since $j-1$ is **011** *in binary, the derivation uses the first defining rule for X'_3, then the second rule for X'_2, and then again the second rule for X'_1 to derive a sentential form containing X'_0. The corresponding derivation is*

$$\mathbf{X'_3} \Rightarrow \mathbf{X'_2}X_2 \Rightarrow X_1\mathbf{X'_1}X_2 \Rightarrow \mathbf{X_1}X_0X'_0X_2 \Rightarrow X_0X_0X_0X'_0X_2 \Rightarrow \ldots$$



The language L_{2^n} is finite and therefore regular. We show that any two words $w_1, w_2 \in \{0,1\}^{2^n}$ with $w_1 \neq w_2$ are inequivalent w.r.t. the Myhill-Nerode relation of L_{2^n} which implies that the minimal DFA for L_{2^n} must have at least 2^{2^n} states. The same argument then shows that the minimal DFA for ∇L_{2^n} must also have at least 2^{2^n} states.

Consider the first position from the right where w_1 and w_2 differ, so w.l.o.g. we have $w_1 = \alpha 0 \beta$ and $w_2 = \alpha' 1 \beta$ for some $\alpha, \alpha', \beta \in \{0,1\}^*$. As a distinguishing word set $v := 1^{2^n - |\beta|} 0 1^{2^n - |\alpha| - 1}$. Note that

$$w_1 v = \alpha 0 \beta 1^{2^n - |\beta|} 0 1^{2^n - |\alpha| - 1} \in L_{2^n},$$

but

$$w_2 v = \alpha' 1 \beta 1^{2^n - |\beta|} 0 1^{2^n - |\alpha| - 1} \notin L_{2^n},$$

and hence any two words w_1 and w_2 in $\{0,1\}^{2^n}$ are inequivalent.

The crucial observation is that from $|w_1 v| = |w_2 v| = 2 \cdot 2^n + 1$ it also follows that $w_1 v \in \nabla L_{2^n}$ and $w_2 v \notin \nabla L_{2^n}$ since the subword closure can only add new words of length at most $2 \cdot 2^n$. This shows that also the minimal DFA for ∇L_{2^n} must have at least 2^{2^n} states. Again, the very same argument works for ΔL_{2^n} since the superword closure can only add words *longer than* $2 \cdot 2^n + 1$. Hence, also the size of the minimal DFA for ΔL_{2^n} is at least doubly-exponential in the size of G_n. The following theorem sums up our discussion.

Theorem 7.11. *There exists a family of CFGs G_n of size $\mathcal{O}(n)$ (generating finite languages) such that the minimal DFAs accepting either $L(G_n)$, or $\nabla L(G_n)$, or $\Delta L(G_n)$, each have at least 2^{2^n} states.*

7.3.3 Regular Expressions

Regular Expressions are another classical representation of regular languages. It is well-known that regular expressions can be exponentially more succinct than DFAs but unfortunately, they can also be exponentially "wasteful" compared to NFAs. More specifically, there are regular languages representable by an NFA of size $\mathcal{O}(n)$ that require a regular expression of size $2^{\Omega(n)}$ [GH08, GH14]. Therefore our results for representing the subword closure via NFAs only yield a very crude doubly exponential upper bound on the size of a regular expression describing the subword closure of a CFG. In this section we show that this can be improved to also yield an upper bound of $2^{\mathcal{O}(|G|)}$ for the size of a regular expression. Note that this new result implies our previous upper bound of $2^{\mathcal{O}(|G|)}$ for the size of an NFA from Section 7.3.1 but this is mainly of theoretical interest since the constant hidden in the $\mathcal{O}$ is larger.

In the following, we view grammars as a system of equations over the idempotent semiring of formal languages $(2^{\Sigma^*}, \cup, \cdot, \emptyset, \{\varepsilon\})$ augmented with the identity $x + 1 = x$. Such idempotent semirings satisfying $x = x + 1$ are called *lossy semirings* in [EKL08]. This identity enforces the languages in our semiring to be subword-closed, for example we have

$$\{ab, bc\} = \{a\} \cdot \{b\} + \{b\} \cdot \{c\} = \{a, \varepsilon\} \cdot \{b, \varepsilon\} + \{b, \varepsilon\} \cdot \{c, \varepsilon\} = \{ab, bc, a, b, c, \varepsilon\}.$$

Consider now a grammar in simple C2F and the defining equation for some nonterminal X_i

$$X_i = \sum_{XY \in Q(X_i)} XY + \sum_{X \in L(X_i)} X + \sum_{C \in C_l(X_i)} CX_i + \sum_{C \in C_r(X_i)} X_i C$$

We can write the solution to this equation succinctly with coefficients $\mu_k, \lambda_{k,l} \in \{0, 1\}$ indicating whether the respective monomial is present in the original equation:

$$X_i = \Sigma_l(X_i)^* \cdot \underbrace{\left(\sum_{1 \leqslant k, l \leqslant (i-1)} \lambda_{k,l} X_k X_l + \sum_{k=1}^{i-1} \mu_k X_k \right)}_{=:\widetilde{X_i}} \cdot \Sigma_r(X_i)^*. \tag{7.1}$$

In $\widetilde{X_i}$ we can factor out each X_k to get

$$\widetilde{X_i} = \sum_{k=1}^{i-1} X_k \cdot \left(\sum_{l=1}^{k} \lambda_{k,l} X_l + \mu_k \right) + \sum_{k=1}^{i-1} \left(\sum_{l=1}^{k-1} \lambda_{k,l} X_l \right) \cdot X_k. \tag{7.2}$$

Now we can use this factorization to compute a regular expression for ∇X_i in a bottom-up fashion as before. As a measure of the size of a regular expression r, we use its *alphabetic width* $\mathrm{alph}(r)$, i.e. the total number of occurrences of elements of Σ in r. Despite the simplicity of $\mathrm{alph}(r)$, all other well-known complexity measures for regular expressions (e.g. number of symbols needed to write down r) can be bounded by a constant times $\mathrm{alph}(r)$ [EKSW04].

Theorem 7.12. *For a grammar* G *in simple C2F with* n *nonterminals, we can find a regular expression* r *representing* $\nabla\mathcal{L}(G)$ *such that* $\text{alph}(r) \leqslant 4 \cdot 5^{n-1}$

Proof. Like in the proof of Theorem 7.7 for NFAs, assume a reverse topological ordering on the nonterminals $X_1, \dots, X_n$ with X_n being the start symbol. We write $r_{\nabla X}$ for the regular expression representing the subword closure of the nonterminal X.

The proof is by induction on n. For $n = 1$ we only have a single rule in the grammar of the form $A_a \to a$ and the regular expression $r_{\nabla A_a} = (a + \varepsilon)$ has size $\text{alph}(a + \varepsilon) = 1$. Let $a_i := \text{alph}(r_{\nabla X_i})$. By induction, we obtain that $a_i \leqslant 4 \cdot 5^{i-1}$ for $i < n$.

By 7.1 and our factorization of the equation for X_n shown in 7.2 we can obtain the upper bound

$$\begin{aligned}
a_n &= \text{alph}(\Sigma_l(X_n)^*) + \text{alph}(\Sigma_r(X_n)^*) + \text{alph}(\widetilde{X_n}) \\
&\leqslant 2(n-1) + \sum_{k=1}^{n-1}\left(a_k + \sum_{l=1}^{k} a_l\right) + \sum_{k=1}^{n-1}\left(a_k + \sum_{l=1}^{k-1} a_l\right) \\
&\overset{\text{IH}}{\leqslant} 2(n-1) + \sum_{k=1}^{n-1}\left(4 \cdot 5^{k-1} + 4 \cdot \sum_{l=1}^{k} 5^{l-1}\right) + \sum_{k=1}^{n-1}\left(4 \cdot 5^{k-1} + 4 \cdot \sum_{l=1}^{k-1} 5^{l-1}\right) \\
&= 2(n-1) + \sum_{k=1}^{n-1}\left(4 \cdot 5^{k-1} + 5^{k} - 1\right) + \sum_{k=1}^{n-1}\left(4 \cdot 5^{k-1} + 5^{k-1} - 1\right) \\
&= 2(n-1) + 9 \cdot \left(\sum_{k=1}^{n-1} 5^{k-1}\right) - (n-1) + 5 \cdot \left(\sum_{k=1}^{n-1} 5^{k-1}\right) - (n-1) \\
&= \frac{9}{4}\left(5^{n-1} - 1\right) + \frac{5}{4}\left(5^{n-1} - 1\right) \leqslant 4 \cdot 5^{n-1}
\end{aligned}$$

□

Corollary 7.13. *For any CFG* G *there is a regular expression* r *of size* $|r| \in 2^{\mathcal{O}(|G|)}$ *representing the subword closure of* G.

Note that the factorization we use above is by no means optimal. Indeed, consider again our example from the beginning of Section 7.3 to illustrate that a straightforward construction produces a representation (NFA or regular expression) of size $2^{|G| \log |G|}$.

$$\begin{aligned}
A_0 &= a \\
A_1 &= A_0A_0 \\
A_2 &= A_0A_0 + A_0A_1 + A_1A_0 + A_1A_1 \\
A_3 &= A_0A_0 + A_0A_1 + A_0A_2 + A_1A_0 + A_1A_1 + A_1A_2 + A_2A_2 \\
&\vdots \\
A_n &= \sum_{0 \leqslant i,j \leqslant (n-1)} A_iA_j.
\end{aligned}$$

This system can be presented more succinctly if we factor the right-hand-sides completely as

$$\begin{aligned}
A_0 &= a \\
A_1 &= A_0 A_0 \\
A_2 &= A_0(A_0 + A_1) + A_1(A_0 + A_1) = (A_0 + A_1) \cdot (A_0 + A_1) \\
&\vdots \\
A_n &= \sum_{0 \leqslant i,j \leqslant (n-1)} A_i A_j = \left(\sum_{i=0}^{n-1} A_i \right) \cdot \left(\sum_{i=0}^{n-1} A_i \right).
\end{aligned}$$

We can view the quadratic monomials of an equation in C2F as a bipartite graph on two disjoint copies of $\mathcal{X}$ and an edge between X_i and X_j if the monomial $X_i X_j$ appears in the equation. In our special case above, the right-hand-sides of the equations form complete bipartite graphs and can thus be factored easily. It is easy to see that the optimal factoring problem reduces to finding a cover of this bipartite graph by a collection of complete bipartite graphs (also called a *biclique cover*). Unfortunately, the minimum biclique cover problem is NP-complete even for chordal bipartite graphs [Orl77, Mü96].

Simple Regular Expressions The restricted class of simple regular expressions (SREs) was propsed in [ACBJ04] as efficient representation for state spaces of lossy channel systems.

Definition 7.14 ([ACBJ04]). *An* atomic expression *is a regular expression over* Σ *of the form*

- $(a + \varepsilon)$, *or*
- $(a_1 + \cdots + a_k)^*$

with $a, a_i \in \Sigma$. *A* product *is a finite product* $p = A_1 \cdots A_n$ *where* A_i *are atomic expressions. A* simple regular expression *(SRE) s is a finite sum of products* P_i, *i.e.* $s = P_1 + \cdots + P_m$.

It is straightforward to see that the language represented by an SRE is subword-closed. Conversely, it is shown in [ACBJ04] that every subword-closed language can be represented by an SRE.

Example 7.15. $s = a^*(b + \varepsilon) + (c + \varepsilon) \cdot (d + \varepsilon)$ *is an SRE while* $r = (a + b)c^*$ *is not.*

The expression $p = (a + b)^* \cdot (b + c + \varepsilon)$ *is not an SRE but can be turned into the equivalent SRE* $p' = (a + b)^* \cdot (c + \varepsilon) + (a + b)^* \cdot (b + \varepsilon)$ *at the expense of increasing its size.*

SREs are an attractive representation since they enjoy good algorithmic propertiese. For example they have a canonical representation and inclusion testing between two SREs is solvable in quadratic time. However, general regular expressions can be exponentially more succinct than SREs:

Remark 7.16. *For $\Sigma = \{a, b\}$ consider the subword-closed language $L_n = \Sigma^{\leqslant n}$. $|L| = 2^{n+1} - 1$*

$$L = \mathcal{L}((a + b + \varepsilon)^n)$$

So L can be represented by a regular expression of size $\mathcal{O}(n)$. An SRE cannot contain a Kleene star since L is a finite language, and hence it has to enumerate the elements of L using products of the atomic expressions $(a + \varepsilon)$ and $(b + \varepsilon)$ so its size is 2^n.

Going further, we see that L_{2^n} can be generated by a CFG of size $\mathcal{O}(n)$, again using the idea of repeated squaring:

$$\begin{aligned} A_n &\to A_{n-1}A_{n-1} \\ A_{n-1} &\to A_{n-2}A_{n-2} \\ &\vdots \\ A_1 &\to A_0A_0 \\ A_0 &\to a \mid b \mid \varepsilon \end{aligned}$$

As remarked, the language can be described by a regular expression (and by an NFA) of size $\mathcal{O}(2^n)$: $L_{2^n} = \mathcal{L}((a + b + \varepsilon)^{2^n})$ whereas its representation as an SRE has size 2^{2^n}. This shows that SREs can be as "wasteful" as DFAs for describing the subword closure of a CFG.

7.4 Application: Approximate Answers to Undecidable Grammar Problems

Suppose we can describe the behaviors of a system (e.g. a recursive program modeled as a pushdown system) using a context-free grammar G_1 and are given a context-free specification G_2 describing e.g. all safe executions. Then the system is safe, if $\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)$, or the system conforms exactly to its specification, if $\mathcal{L}(G_1) = \mathcal{L}(G_2)$. More generally, linear-time model-checking corresponds to language inclusion problems as remarked in [KPV02].

It is well-known that inclusion or equivalence checking of context-free grammars are undecidable problems, and hence, we want to develop approximation methods to tackle these problems and to obtain practically efficient semi-decision procedures.

Given context-free grammars G_1, G_2 let $L_1 = \mathcal{L}(G_1)$ and $L_2 = \mathcal{L}(G_2)$. Since $L_1 = L_2$ implies that $\nabla L_1 = \nabla L_2$ we can use the subword (or superword) closure to obtain counterexamples to equivalence: $\nabla L_1 \neq \nabla L_2 \implies L_1 \neq L_2$. Accordingly, $L_1 \subseteq L_2$ implies $\nabla L_1 \subseteq \nabla L_2$ so we can also use the approach for checking non-inclusion between CFLs.

It is important to note that this approach is only justified since the subword closure only depends on the language and *not* on its presentation (i.e. the grammar). Other methods, like the Mohri-Nederhof approximation [MN01] compute a regular over-approximation of L that depends on the grammar. We still can use such approximations to check inequivalence between L_1 and L_2 in a two step approach: First over-approximate L_1 and L_2 by regular languages R_1 and R_2, respectively. Then check whether both inclusions $L_1 \subseteq R_2$ and $L_2 \subseteq R_1$ hold (note that these are decidable problems). If one of them does not hold then surely $L_1 \neq L_2$.

First, we only consider the approach based on sub-/superword closure approximation and focus on language inequality checking. Our approach is shown in Algorithm 11.

Algorithm 11: High-level outline of inequivalence checking via subword closure approximation.

```
input : CFGs $G_1, G_2$
output: Witness $w \in \mathcal{L}(G_1) \oplus \mathcal{L}(G_2)$ or "maybe equal".

while timeout not reached do
    $R_1 \leftarrow \nabla\mathcal{L}(G_1)$;   /* $R_i$: any representation (DFA,NFA,SRE) */
    $R_2 \leftarrow \nabla\mathcal{L}(G_2)$
    if $\mathcal{L}(R_1) \neq \mathcal{L}(R_2)$ then
        return generateWitness($G_1$,$G_2$,$R_1$,$R_2$)
    else
        $G_1, G_2 \leftarrow$refine($G_1$,$G_2$)
    end
end
return "maybe equal"
```

A refined implementation of the above outlined algorithm has to state explicitly how to

- represent the subword closures $R_i = \nabla\mathcal{L}(G_i)$,
- solve the equivalence problem $\mathcal{L}(R_1) \overset{?}{=} \mathcal{L}(R_2)$,
- generate witnesses if $\mathcal{L}(R_1) \neq \mathcal{L}(R_2)$, and
- refine the problem if $\mathcal{L}(R_1) = \mathcal{L}(R_2$.

7.4.1 Equivalence Checking of NFAs Modulo Closure

It is well-known that the universality and equivalence problem for NFAs is PSPACE-complete. Even more, the problem stays PSPACE-complete for prefix-, or suffix-, or factor-closed languages as shown in [KRS09, RSX12]. In all cases, PSPACE hardness is established via reduction from the PSPACE-complete universality problem, i.e. to decide whether $\mathcal{L}(\mathcal{A}) = \Sigma^*$ for an NFA $\mathcal{A}$ [Hun73, MS72].

However, for both subword and superword closed languages the universality problem is solvable in linear time:

- For subword-closed L, $L = \Sigma^*$ if and only if there is an SCC in the NFA such that all letters from Σ occur in it [RSX12].
- For superword-closed L, $L = \Sigma^*$ if and only if $\varepsilon \in L$.

This means we can check universality for both types of languages in linear time in the size of the NFA and hence, the usual hardness proof for equivalence is not applicable to neither subword- nor superword-closed languages. A natural question is therefore whether the equivalence problem for NFAs is easier for sub-/superword-closed languages, and indeed we prove that equivalence testing of such NFAs is only coNP-complete.

Let $\mathsf{cl} : 2^{\Sigma^*} \to 2^{\Sigma^*}$ be a (computable) function on languages. We call two NFAs $\mathcal{A}_1, \mathcal{A}_2$ *equivalent modulo* cl, written as $\mathcal{A}_1 \equiv_{\mathsf{cl}} \mathcal{A}_2$, if $\mathsf{cl}(\mathcal{L}(\mathcal{A}_1)) = \mathsf{cl}(\mathcal{L}(\mathcal{A}_2))$. Let us write $\mathcal{A}_1 \overset{?}{\equiv}_{\mathsf{cl}} \mathcal{A}_2$ for the corresponding decision problem: given two NFAs $\mathcal{A}_1, \mathcal{A}_2$, decide whether $\mathcal{A}_1 \equiv_{\mathsf{cl}} \mathcal{A}_2$. Phrased in this language, the results in [KRS09, RSX12] state that $\mathcal{A}_1 \overset{?}{\equiv}_{\mathsf{cl}} \mathcal{A}_2$ is PSPACE-complete for cl being the prefix, suffix, or factor closure operation.

To prove that $\mathcal{A}_1 \overset{?}{\equiv}_{\mathsf{cl}} \mathcal{A}_2$ is solvable in coNP for $\mathsf{cl} = \nabla$, we show that if $\mathcal{A}_1 \not\equiv_{\nabla} \mathcal{A}_2$ then there exists a word w of polynomial length in the size of $|\mathcal{A}_1| + |\mathcal{A}_2|$ such that $w \in \mathcal{L}(\mathcal{A}_1) \oplus \mathcal{L}(\mathcal{A}_2)$, where $\oplus$ denotes the symmetric difference of sets.

The next lemma allows us to assume (w.l.o.g.) that an NFA for a sub-/superword-closed language has a particular structure.

Lemma 7.17. *Let $\mathcal{A}$ be an NFA. Define $\mathcal{A}^{\nabla}$ as the NFA we obtain from $\mathcal{A}$ by adding for every transition $q \xrightarrow{a} q'$ of $\mathcal{A}$ the ε-transition $q \xrightarrow{\varepsilon} q'$.*

Similarly, define $\mathcal{A}^{\Delta}$ as the NFA we obtain by adding the self-loops $q \xrightarrow{a} q$ for every state q and every terminal $a \in \Sigma$ to $\mathcal{A}$. Then $\nabla\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^{\nabla})$ and $\Delta\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^{\Delta})$.

Proof. We start with $\nabla\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^{\nabla})$: Pick any $w \in \nabla\mathcal{L}(\mathcal{A})$. Then there is some $w' \succcurlyeq w$ such that $w' \in \mathcal{L}(\mathcal{A})$, and thus by construction also $w' \in \mathcal{L}(\mathcal{A}^{\nabla})$. That is there is an accepting run $q_0 \xrightarrow{x_0} q_1 \xrightarrow{x_1} \dots \xrightarrow{x_l} q_{l+1}$ with $q_{l+1} \in F$ and $w' = x_0 x_1 \dots x_l$ (with potentially $x_i = \varepsilon$ for some i). Using the additional ε-transitions of $\mathcal{A}^{\nabla}$ we therefore can turn this sequence into an accepting sequence for w by simply replacing those x_i by ε which do not occur in w. For the other direction, one can reverse this argument by recalling that for any ε-transition $q \xrightarrow{\varepsilon} q'$ added to $\mathcal{A}^{\nabla}$ there is some $a \in \Sigma$ such that $q \xrightarrow{a} q'$ is a transition of $\mathcal{A}$.

Consider now the second claim $\Delta\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^{\Delta})$: Choose some $w \in \Delta\mathcal{L}(\mathcal{A})$. Then there is some $w' \preccurlyeq w$ such that $w' \in \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}^{\Delta})$. Any accepting run $q_0 \xrightarrow{x_0} q_1 \xrightarrow{x_1} \dots \xrightarrow{x_l} q_{l+1}$ (with $q_{l+1} \in F$ and $w' = x_0 x_1 \dots x_l$) of $\mathcal{A}^{\Delta}$ can then be extended to an accepting run of $\mathcal{A}^{\Delta}$ for w by using the additional loops of $\mathcal{A}^{\Delta}$ to consume any letters occurring exclusively in w. In the other direction given an accepting run of $\mathcal{A}^{\Delta}$ we simply strip it by any loops which is guaranteed to yield an accepting run (for a scattered subword) of $\mathcal{A}$ as the transition relations of $\mathcal{A}$ and $\mathcal{A}^{\Delta}$ only differ in loops. □

Note that the DFA resulting from the powerset construction applied to such an NFA can be of superpolynomial size as shown by Okhotin [Okh10], but is has a particularly simple structure (which has also been observed in [Okh10]):

Lemma 7.18. *Let $\mathcal{A}$ be an NFA. Let $\mathcal{D}^{\nabla}_{\mathcal{A}}$ (resp. $\mathcal{D}^{\Delta}_{\mathcal{A}}$) be the DFA we obtain from $\mathcal{A}^{\nabla}$ (resp. $\mathcal{A}^{\Delta}$) via the powerset construction. For any transition $S \xrightarrow{a} T$ of $\mathcal{D}^{\nabla}_{\mathcal{A}}$ ($\mathcal{D}^{\Delta}_{\mathcal{A}}$) it holds that $S \supseteq T$ (resp. $S \subseteq T$).*

Proof. Recall that the state (sets) of $\mathcal{D}^{\nabla}_{\mathcal{A}}$ are closed w.r.t. taking ε-successors in $\mathcal{A}^{\nabla}$. As $\mathcal{A}^{\nabla}$ was obtained from $\mathcal{A}$ by introducing for every transition $q \xrightarrow{a} q'$ ($a \in \Sigma$) the ε-transition $q \xrightarrow{\varepsilon} q'$, this means that, if $q \in S$, then every state reachable from q in the directed graph underlying $\mathcal{A}$ has to be included in S, too. As for any transition $S \xrightarrow{a} T$ in $\mathcal{D}^{\nabla}_{\mathcal{A}}$, T is a subset of the states reachable from S, the claim follows.

In case of the superword closure, pick any transition $S \xrightarrow{a} T$ of $\mathcal{D}^{\Delta}_{\mathcal{A}}$ and any state $q \in S$. Then by construction of $\mathcal{A}^{\Delta}$ there is the loop $q \xrightarrow{a} q$ in $\mathcal{A}^{\Delta}$ which implies that also $q \in T$ by definition of the powerset construction. □

By this result, the transition relation of $\mathcal{D}^{\nabla}_{\mathcal{A}}$ (disregarding self-loops) can be "embedded" into the lattice of subsets of the states of $\mathcal{A}$, which has height $|\mathcal{A}|$.

Corollary 7.19. *With the assumptions of the preceding lemma: The length of the longest simple path in $\mathcal{D}^{\nabla}_{\mathcal{A}}$ (resp. $\mathcal{D}^{\Delta}_{\mathcal{A}}$) is at most $|\mathcal{A}|$.*

It now immediately follows that a shortest separating word for sub- resp. superword closed NFAs – if one exists – has at most length linear in the size of the two NFAs.

Lemma 7.20. *Let $\mathcal{A}$ and $\mathcal{B}$ be two NFAs. If $\mathcal{A} \not\equiv_{\nabla} \mathcal{B}$ (resp. $\mathcal{A} \not\equiv_{\Delta} \mathcal{B}$), then there exists a word $w \in \nabla\mathcal{L}(\mathcal{A}) \oplus \nabla\mathcal{L}(\mathcal{B})$ (resp. $w \in \Delta\mathcal{L}(\mathcal{A}) \oplus \Delta\mathcal{L}(\mathcal{B})$) of length at most $|\mathcal{A}| + |\mathcal{B}|$.*

Proof. Assume $\mathcal{A} \not\equiv_{\nabla} \mathcal{B}$, and let w be a shortest separating word. Consider the unique run of the product DFA $\mathcal{D}^{\nabla}_{\mathcal{A}} \times \mathcal{D}^{\nabla}_{\mathcal{B}}$ on $w = w_0 w_1 \dots w_l$:

$$(L_0, R_0) \xrightarrow{w_0} (L_1, R_1) \xrightarrow{w_1} \dots \xrightarrow{w_l} (L_l, R_l).$$

By the preceding lemma we then have $L_i \supseteq L_{i+1}$ and $R_i \supseteq R_{i+1}$ along the run. As w is assumed to be a shortest separating word, it has to hold that $\neg(L_i = L_{i+1} \wedge R_i = R_{i+1})$ for all $i = 1, \dots, l-1$. In other words, we have

$$|\mathcal{A}| + |\mathcal{B}| \geqslant |L_0| + |R_0| > |L_1| + |R_1| > \dots > |L_l| + |R_l| \geqslant 1$$

from which the claim immediately follows.

In the case of the superword closure one deduces in the same way that the accepting run for a shortest separating word has to satisfy:

$$1 \leqslant |L_0| + |R_0| < |L_1| + |R_1| < \ldots < |L_l| + |R_l| \leqslant |\mathcal{A}| + |\mathcal{B}|$$

□

Theorem 7.21. *The decision problems $\mathcal{A} \overset{?}{\equiv}_\nabla \mathcal{B}$ and $\mathcal{A} \overset{?}{\equiv}_\Delta \mathcal{B}$ are in* coNP.

Proof. By the preceding lemma, if $\mathcal{A} \not\equiv_\nabla \mathcal{B}$ there exists a witness w in the symmetric difference of the two closures, $w \in \nabla\mathcal{L}(\mathcal{A}) \oplus \nabla\mathcal{L}(\mathcal{B})$ such that $|w| \leqslant |\mathcal{A}| + |\mathcal{B}|$. Now we can guess w in nondeterministic polynomial time and hence the in-equivalence problem $\mathcal{A} \overset{?}{\not\equiv}_\nabla \mathcal{B}$ is in NP. The reasoning for $\mathcal{A} \overset{?}{\equiv}_\Delta \mathcal{B}$ is exactly the same. □

To show coNP-hardness of the equivalence problem we use a standard reduction from the coNP-complete problem TAUT (to decide validity for formulæ in propositional calculus). The strategy is essentially the same as the well-known coNP-hardness proof for the equivalence problem of regular expressions without Kleene stars.

Theorem 7.22. *The decision problems $\mathcal{A}_1 \overset{?}{\equiv}_\nabla \mathcal{A}_2$ and $\mathcal{A}_1 \overset{?}{\equiv}_\Delta \mathcal{A}_2$ are* coNP*-hard.*

Proof. Let φ be a formula of propositional calculus, (w.l.o.g.) in disjunctive normal form. We construct a regular expression which encodes all satisfying assignments of φ:

Let $x_1, x_2, \ldots, x_n$ be the propositional variables occurring in φ, and assume that $\varphi = \bigvee_{i\in[k]} C_i$ with $C_i = \bigwedge_{j\in[l_i} L_{i,j}$ and $L_{i,j}$ literals. Further, we may assume that in every conjunction C_i is contradiction free. We associate with every C_i a simple regular expression ρ_i enumerating all satisfying assignments of D_i: Initially, set $\rho_i = \emptyset$. Going from $j = 1$ to $j = n$, if x_j occurs in C_i, then set $\rho_i := \rho_i 1$; if $\neg x_j$ occurs in C_i, set $\rho_i := \rho_i 0$; otherwise set $\rho_i := \rho_i(0 + 1)$. Finally, set $\rho := \rho_1 + \rho_2 + \ldots + \rho_k$. Obviously, the size of ρ is polynomial in the size of φ. Further, we can compute an NFA $\mathcal{A}$ from ρ in time polynomial in $|\rho|$, such that $\mathcal{L}(\rho) = \mathcal{L}(\mathcal{A})$. Note that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\rho) \subseteq \Sigma^n$ by construction. In particular, $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\rho) = \Sigma^n$ if and only if φ is a tautology.

It therefore suffices to show that $\nabla\mathcal{L}(\mathcal{A}) = \Sigma^{\leqslant n}$ (resp. $\Delta\mathcal{L}(\mathcal{A}) = \Sigma^{\geqslant n}$) if and only if $\mathcal{L}(\mathcal{A}) = \Sigma^n$. But this is easy as the subword closure resp. superword closure can only add words of length less resp. greater than n. □

7.4.2 Witness Generation

In case Algorithm 11 discovers that $\mathcal{L}(G_1) \not\equiv_\nabla \mathcal{L}(G_2)$ is has found a word in the symmetric difference of the approximations $w \in \nabla\mathcal{L}(G_1) \oplus \nabla\mathcal{L}(G_2)$ which is an indirect witness that $\mathcal{L}(G_1) \neq \mathcal{L}(G_2)$. However, w is not suitable to verify directly that the two languages are different. It would be much more convincing if the algorithm outputted a word $w' \in \mathcal{L}(G_1) \oplus \mathcal{L}(G_2)$. Here we show several ways to do this. For simplicity we assume in the following (w.l.o.g.) that $w \in \nabla\mathcal{L}(G_1)$ and $w \notin \nabla\mathcal{L}(G_2)$.

One way to find a suitable witness $w' \in \mathcal{L}(G_1) \setminus \mathcal{L}(G_2)$ given a word $w \in \nabla\mathcal{L}(G_1) \setminus \nabla\mathcal{L}(G_2)$ is the following: Assuming (w.l.o.g.) that G_1 and G_2 contain no ε-productions produce a grammar G_1' that generates $\nabla\mathcal{L}(G_1)$ by adding for each nonterminal A the production $A \to \varepsilon$. Then parse the word w using the grammars G_1'. For each (previously added) rule $A \to \varepsilon$ that is used in the parse, replace the corresponding ε-part in w by the shortest word derivable from A in G_1. This whole procedure yields a (possibly long) superword w' of w for which we cannot derive a good upper bound on its length.

In practice we use a different approach that has the potential to produce shorter witnesses w' is to generate a shortest word in $\mathcal{L}(G_1) \setminus \nabla\mathcal{L}(G_2)$: compute the DFA $\mathcal{D}_c$ for $\overline{\mathcal{L}(G_2)}$, intersect it with G_1 and produce a shortest word in the resulting CFG. Since $|\mathcal{D}_c| \in 2^{2^{\mathcal{O}(|G_2|)}}$, the size of the intersection grammar is polynomial in the size of $\mathcal{D}_c$ and $|G_1|$, and since a shortest word of a grammar G is at most exponential in $|G|$, we get in total a (very crude) triple exponential bound on the length of w' (in $|G_1| + |G_2|$). However, according to our experiments (cf. Section 7.4.4) this worst-case behavior does not seem to occur in practice.

7.4.3 Refinement

If Algorithm 11 discovers that $\mathcal{L}(R_1) = \mathcal{L}(R_2)$, then we can neither conclude that $G_1 \equiv G_2$ nor that $G_1 \not\equiv G_2$. To turn our algorithm into an abstraction-refinement scheme we need means to refine our subword-approximations. A strategy we considered is refinement by language covering: We cover $\mathcal{L}(R_1)$ (which is equal to $\mathcal{L}(R_2)$) by a finite collection of regular languages $\mathcal{L}(R_1) \subseteq L' := L_0 \cup L_1 \cup \cdots \cup L_k$ and branch our search for counterexamples to $\mathcal{L}(G_1) \overset{?}{=} \mathcal{L}(G_2)$ into $k+1$ subproblems of the form $\mathcal{L}(G_1) \cap L_i \overset{?}{=} \mathcal{L}(G_2) \cap L_i$. Since L' covers $\mathcal{L}(R_1)$, we do not cut off potential counterexamples. The problem of this search tree method is to find the right balance between breadth and depth: avoiding to generate too many subproblems or having to refine too many times. For the subword-approximation to be useful on the subproblems, we want the languages L_i to be infinite (except maybe for L_0 which can then be checked via enumeration).

One instance of this covering strategy is *prefix-refinement*. Here we consider all the words $p_1, \ldots, p_k \in \Sigma^d \cap \mathcal{L}(R_1)$ with some small length d called the *depth* of the refinement and define the languages L_i by $L_0 := \nabla\{p_1, \ldots, p_k\}$, and $L_i := p_i\Sigma^*$ for $i \in [k]$. Note that L_0 is a finite set and hence we could solve the problem $\mathcal{L}(G_1) \cap L_0 \overset{?}{=} \mathcal{L}(G_2) \cap L_0$ by enumeration. For all other subproblems we simply iterate our abstraction loop. Note that considering prefixes of length d of words in $\mathcal{L}(R_1)$ is usually much more efficient than just using all words in Σ^d. Furthermore, the words p_i can easily be generated by inspecting the finite automaton for $\mathcal{L}(R_1)$.

It is easy to see that using the prefix refinement, we obtain a semi-decision procedure for CFG inequivalence, i.e. if there exists a $w \in \mathcal{L}(G_1) \oplus \mathcal{L}(G_2)$, then our method eventually reports *some* counterexample $w' \in \mathcal{L}(G_1) \oplus \mathcal{L}(G_2)$. Note that this is not a strong result – exhaustive enumeration of words in Σ^* also yields a semi-decision procedure, but as our experiments show our method has the potential to find even long counterexamples quickly (which can be infeasible to find using enumeration).

7.4.4 Experiments

To evaluate the power of our approximation alone we disregard the finite subproblem $\mathcal{L}(G_1) \cap L_0 \overset{?}{=} \mathcal{L}(G_2) \cap L_0$ in our experiments. With this change our method no longer is a semi-decision procedure but this can be easily repaired by combining it with an enumeration approach for equivalence testing, like [AHL08].

The paper [AHL08] presents CFGANALYZER, a tool that uses SAT-solving to attack several undecidable grammar problems by exhaustive enumeration. We demonstrate the feasibility of our approximation approach on several slightly altered grammars (cf. [VT13]) for the PASCAL programming language[3]. The altered grammars were obtained by adding, deleting, or mutating a single rule from the original grammar [VT13]. We used FPSOLVE and CFGANALYZER to check equivalence of the altered grammar with the original. Both tools were given a timeout of 30 seconds.

Note that our method is not designed to replace enumeration-based tools like CFGANALYZER, we rather envision a combined approach: Use overapproximations like the subword closure (with small refinement depth) as a quick check and resort to more computationally demanding techniques like SAT-solving for a thorough test. Also note that it is not too hard to find examples where enumeration-based tools cannot detect inequivalence anymore, e.g. by considering grammars with large alphabet (like C# or Java) for which the shortest word in the language is already longer than 20 tokens. Here we just present a case study where both approaches can be fruitfully combined.

Table 7.1 demonstrates that even if our tool uses the very simple prefix-refinement (which is the main bottleneck in terms of speed), we can successfully solve 100 cases where CFGANALYZER has to give up after 30 seconds and even in cases where both tools find a difference, FPSOLVE does so much faster.

7.5 Related Work and Conclusions

The subword closure has been studied previously as an abstraction in verification. In [ACBJ04] the authors present a symbolic method for reachability analysis of lossy channel systems, where the set of states of the system is a subword-closed language. [ACBJ04] proposed simple regular expressions (SREs) to represent subword-closed languages since SREs enjoy several good algorithmic properties (e.g. equivalence testing in quadratic time). Our example from Section 7.3.3 suggests that finite languages are a particular challenge to represent via SREs, similar as for DFAs. We have only briefly

[3]Available from `https://github.com/nvasudevan/experiment/tree/master/grammars/mutlang/acc.`

scenario	# instances	# CA	t_{CA}	#FP	t_{FP}	$\#(CF \wedge FP)$	$t^{\wedge}_{CA}$	$t^{\wedge}_{FP}$
add	700	190	17.9	18	2.43	8	10.7	4.97
delete	284	61	17.8	34	0.424	10	14.4	0.464
empty	69	32	18.7	1	1.35	1	5.62	1.35
mutate	700	167	19.1	100	1.3	36	15.8	2.87
switchadj	187	16	20.5	2	5.46	1	9.68	0.34
switchany	328	35	18	9	3.72	8	9.09	2.84
$\sum$	2268	501	–	164	–	64	–	–

Table 7.1: Numbers of solved instances for different scenarios and respective average times: #CA: solved by CFGANALYZER, #FP: solved by FPSOLVE, $\#(CA \wedge FP)$: solved by both tools, $t^{\wedge}_{tool}$: time needed by *tool* on instances from $(CA \wedge FP)$.

touched the descriptional complexity of the subword closure using SREs. It would be interesting to compare their succinctness to DFAs. We leave open to study whether SREs are a suitable representation for the subword closure of CFGs in practice – their performance in verification problems suggests that they are more suitable than e.g. DFAs [ACBJ04].

Recently, Karandikar, Niewerth, and Schnoebelen have used and extended our results to study the state complexity of the subword *interior* of regular languages [KS14, KNS14]. The subword interior of a language $L \subseteq \Sigma^*$ is defined as

$$\mathrm{subint}(L) := \{w \in \Sigma^* : \nabla w \subseteq L\},$$

or equivalently, as the complement of the superword closure of the complement of L, i.e.

$$\mathrm{subint}(L) = \Sigma^* \setminus \Delta(\Sigma^* \setminus L).$$

The subword interior is the largest subword-closed set that is included in a language and can be viewed as an underapproximation. However, for CFGs the subword interior is not effectively computable: If it were we could decide universality of a CFL L using

$$L = \Sigma^* \Leftrightarrow \Sigma^* \setminus L = \emptyset \Leftrightarrow \Delta(\Sigma^* \setminus L) = \emptyset \Leftrightarrow \mathrm{subint}(L) = \Sigma^*.$$

8 Application III: Computational Linguistics and Natural Language Processing

In this chapter we present an application of the notion of tree dimension (aka. Strahler number) to the field of computational linguistics and natural language processing. In particular we make two contributions which are published in [SLE14].

1. We argue that tree dimension is a reasonable measure of sentence complexity and investigate empirically the dimension-distribution of several natural languages.
2. We outline an application to natural language processing . Specifically, we propose to use dimension as annotation to improve statistical parsing and compare it to other annotation methods.

In Section 8.1 we study the dimension of parse trees of natural language in detail. Using the combinatorial properties of tree dimension, we argue that dimension is a meaningful measure of sentence complexity and we empirically investigate the dimension distributions for several treebanks (databases of human generated parse trees for natural language texts).

In Section 8.2 we first recall some basics of grammar-based parsing. We then review well-known techniques for improving parsing accuracy by refining grammars through annotation.

We investigate in Section 8.3 how to incorporate dimension information into parsers trained from treebanks. As a case study we use the Tübingen treebank of written German (TüBa-D/Z) and experimentally demonstrate that our annotation significantly improves both parsing accuracy and parsing speed. Moreover, we show that our heuristic can be fruitfully combined with previous techniques.

We conclude and outline some promising paths for future work in Section 8.4

All results of this chapter have appeared in [SLE14]. Here we supplement our results with some more context, explanations, and examples.

8.1 Tree Dimension as a Measure of Sentence Complexity

A classical theory in experimental psychology on the capacity of human memory was proposed by George A. Miller [Mil56] and is now called "Miller's Law". Miller proposes that human working memory can only retain a handful of items at the same time. In his work he stressed that it is the number of *items* that is important not the amount of information in bits. In psycholinguistics, his work has stimulated the search for models of human sentence processing that only require bounded memory. For an introductory survey and a comparison of several parsing algorithms w.r.t. their psychological plausibility see [Cro99].

Recall from Section 3.6.3 that $\dim(t)+1$ is the minimum amount of stack-space needed by any top-down traversal of a binary tree t. Hence, tree dimension can be regarded as a (very rough) measure for the structural complexity of a sentence which disregards the order in which subtrees of a parse tree are visited (resp. generated or parsed). In this sense, dimension might be more adequate for describing the complexity of written rather than spoken language.

We propose tree dimension as a new measure of syntactic complexity and empirically show that parse trees have dimension at most 4 for a variety of natural languages.

Example 8.1. *The following are two examples of sentences from natural language that have a (human-generated) parse tree of dimension* 4 *(not shown due to its size):*

"That commercial – which said Mr. Coleman wanted to take away the right of abortion "even in cases of rape and incest", a charge Mr. Coleman denies – changed the dynamics of the campaign, transforming it, at least in part, into a referendum on abortion." — 58 tokens, from: The Wall Street Journal, Penn Treebank (sample shipped with NLTK*).*

"Aber anstatt anzuerkennen, wie genau die Nato bislang die Ziele traf, die sie auch treffen wollte, wie stark, verglichen mit dem Irak-Krieg, die Bemühungen sind, die Zivilisten zu schonen, breitet sich nun Entsetzen aus." — 42 tokens, from: German newspaper "taz", TüBa-D/Z treebank.

The next sentence is of similar length as the first one, but has a parse tree of dimension 2.

"According to reports carried by various news services, the Brazilian government told its sugar producers that they won't be allowed to export sugar during the current 1989-90 season, which began May 1, and the 1990-91 season so that it can be used to produce alcohol for automobile fuel." — 58 tokens, from: The Wall Street Journal, Penn Treebank (sample shipped with NLTK*).*

The previous examples suggest that a sentence with a parse tree of high dimension is complex because it comprises many dependent sub-clauses that have to be remembered while reading it. Miller's Law suggests that parse trees for naturally occurring sentences of human language have small dimension.

We confirm this hypothesis experimentally for several publicly available treebanks (datasets of parse trees generated by human linguists). For comparison we computed the average and maximum dimension and height of all parse trees in a treebank as reported in Table 8.1.

Language	Source	Avg. Dim.	Max. Dim.	Avg. Height	Max. Height
Basque	SPMRL‡	2.12	3	6.57	13
English	Penn♣	2.38	4	10.13	29
French	SPMRL	2.29	4	8.5	34
German	SPMRL	1.94	4	5.26	12
German	TüBa-D/Z♠	2.13	4	7.97	24
Hebrew	SPMRL	2.44	4	11.67	31
Hungarian	SPMRL	2.11	4	6.93	16
Korean	SPMRL	2.18	4	9.9	19
Polish	SPMRL	1.68	3	8.97	22
Swedish	SPMRL	1.83	4	6.7	24

Table 8.1: Average and maximum dimension and height of parse trees for several treebanks of natural languages. ‡: SPMRL dataset [STK$^+$13], ♣: 10% sample from the Penn treebank shipped with NLTK, ♠: TüBa-D/Z treebank [THK$^+$03].

The average dimension of parse trees ranges from 1.68 to 2.44 and the largest dimension we ever encountered was 4. By contrast, the distribution of the height shows a much larger variability. Note that nothing prevents us in principle from generating sentences that have parse trees of dimension 5 or 6 (even larger dimensions would enforce quite long sentences). However, such sentences do not seem to occur in commonly used language and even sentences of dimension 4 are rare.

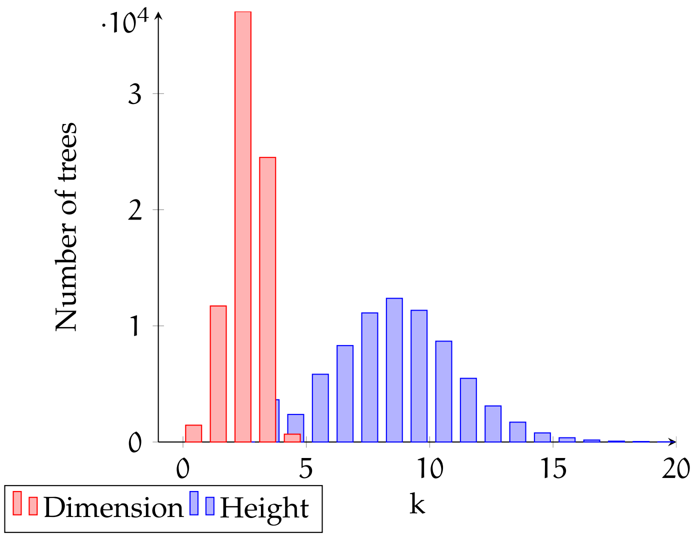


Figure 8.1: Distribution of the dimension and height of all parse trees from the TüBa-D/Z treebank.

To illustrate the distribution of the dimension and height we show a histogram for the TüBa-D/Z treebank in Figure 8.1. These distributions look fairly similar for all languages.

8.2 Introduction to PCFG Parsing

Here, we briefly review standard techniques for parsing with probabilistic context-free grammars (PCFGs). For a more detailed introduction we refer to [MS99].

8.2.1 Motivation

The goal of parsing is to produce a syntactic analysis of an input sentence given as a sequence of words. Parsing is often used as a sub-procedure in more complex NLP tasks like machine translation or document summarization [YK01, KM02, UNMT06].

Modern machine translation systems do not translate sentences word-by-word but try to model the translation as a transformation of syntax trees (formally, as a probabilistic tree transducer or a tree-to-string transducer). To train statistical models one uses a parallel corpus of sentences, e.g. speeches given in the European Parliament translated into different languages by human translators. Such a corpus usually comprises raw sentences not annotated with a parse tree structure. Hence the sentences first have to be parsed into trees to train a statistical model of tree transformations

Document summarization is the task of producing a fixed-length summary of a text which comprises the gist of the content and is grammatically correct. One general approach (text compression), tries to extract a summary by dropping “unimportant” parts of the original text. In order to obtain a grammatically correct summary one cannot just drop arbitrary words. Thus one possibility is to cut “unimportant” subtrees from a parse tree of a sentence. This requires an accurate descriptions of the syntactic structure of a sentence to obtain good results.

8.2.2 PCFGs

To define the parsing task formally, we assume a (usually infinite) set S of sentences and a (likewise infinite) set of syntactic analyses T. Furthermore, we assume a (possibly partial) “yield” function, mapping $t \in T$ to S (if t does not correspond to a sentence this function is undefined). In a formal setting we might have $S \subseteq \Sigma^*$ and T being the set of all derivation trees of a fixed context-free grammar.

To quantify the notion of a “best” parse, we usually assign probabilities to syntactic analyses. There are two possibilities:

- A *generative* model defines a *joint* probability distribution $\Pr[s, t]$ over the set of all sentences s and syntactic analyses t.
- A *discriminative* model only defines a family of *conditional* distributions $\Pr[t \mid s]$ on the set of all parses t, given a sentence s.

The advantage of discriminative models is that they need not model the distribution $\Pr[s]$ which can be complicated. The advantage of generative models is that we can also use them to generate random analyses t (and hence also random sentences). Moreover, we can turn any generative model into a discriminative one by marginalizing over t (which can be an expensive calculation)

$$\Pr[t \mid s] = \frac{\Pr[t, s]}{\Pr[s]} = \frac{\Pr[t, s]}{\sum_{t \in T} \Pr[t, s]}.$$

A probabilistic context-free grammar (PCFG) is a CFG $(\mathcal{X}, \Sigma, \to, S)$ with a designated start symbol S where each rule $r = (X, \alpha) \in \to$ is assigned a probability $p(r) \in [0, 1]$ such that for each nonterminal X we obtain a discrete probability distribution over the rules with left hand side X:

$$\sum_{\alpha : X \to \alpha} p(X, \alpha) = 1 \quad \text{for all } X \in \mathcal{X}$$

The probability of any leftmost derivation using the rules of the grammar

$$S \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_k \Rightarrow w \in \Sigma^*$$

is defined as the product of the probabilities of all rules used. The probability of a derivation tree t is defined as the probability of the unique leftmost derivation described by t.

A PCFG G is called *consistent* if it defines a probability distribution over the set $T := \mathcal{T}_S^G$ of all its derivation trees, i.e.

$$\sum_{t \in T} \Pr[t] = 1.$$

Consistent PCFGs are generative models that define a joint distribution over T and S in the obvious way: If t yields the sentence s then $\Pr[t, s] = \Pr[t]$, otherwise $\Pr[t, s] = 0$. More precisely, we define for $t \in T$ and $s \in \Sigma^*$

$$\Pr[t, s] = \begin{cases} \Pr[t], & \text{if } Y(t) = s \\ 0, & \text{otherwise} \end{cases}$$

This joint distribution also induces a probability distribution on sentences $s \in \Sigma^*$ via

$$\Pr[s] = \sum_{t \in T} \Pr[s, t] = \sum_{t \in T : Y(t) = s} \Pr[t].$$

Given a probabilistic model, parsing is the task of finding the most probable syntactic structure t for a given input sentence s:

$$\arg\max\{\Pr[t \mid s] : t \in T\}.$$

For a PCFG, this optimization problem can be efficiently solved via the Viterbi algorithm which is a "weighted" version of the well-known CYK algorithm for parsing with a CFG. In fact, one can generalize the CYK algorithm (and other parsing algorithms) to a semiring-weighted setting (see [Goo98] for an extensive treatment). In this framework the Viterbi algorithm can be seen as the CYK algorithm instantiated over the "Viterbi semiring" $([0, 1], \max, \cdot, 0, 1)$. For simplicity, we assume the PCFG to be in Chomsky normal form (CNF). We note that the CYK algorithm can also be generalized to grammars in 2NF [LL09]. The basic version, shown in Algorithm 12, only computes the probability of a most-probable parse. However, we can reconstruct the parse-tree by additionally storing back-pointers and performing a singe traceback after filling the chart. From the description, we can see that the algorithm requires $\mathcal{O}(n^3 \cdot |G|)$ operations and needs $\mathcal{O}(n^2 \cdot |G|)$ space.

8.2.3 Inducing Grammars from Treebanks

Treebanks A *treebank* consists of a set of sentences and their parse trees, which are produced by humans (with tool support). Popular treebanks often comprise newspaper texts, such as the

1. Penn Treebank (English, ~ 40.000 sentences from the "Wall Street Journal").
2. TIGER treebank (German, ~ 50.000 sentences from the "Frankfurter Rundschau").
3. TüBa-D/Z treebank of written German (> 80.000 sentences from the "taz").

Algorithm 12: Viterbi algorithm for PCFG parsing – a generalization of CYK.

```
input  : PCFG G, sentence $s \in \Sigma^n$.
output: Probability of the most-probable parse tree, $\max\{\Pr[t \mid s] : t \in T\}$
/* Assume initialization $X_{i,j}(A) = 0$ for all $i, j \in [n], A \in \mathcal{X}$      */
for $i = 1 \ldots n$ do
    foreach rule $A \to s_i$ do
        $X_{i,1}(A) := \Pr[A \to s_i]$
    end
end
for $j = 2 \ldots n$ do
    for $i = 1 \ldots (n - j + 1)$ do
        for $k = 1 \ldots (j - 1)$ do
            foreach rule $r = (A, BC)$ s.t. $X_{i,k}(B) > 0$ and $X_{k,j}(C) > 0$ do
                $X_{i,j}(A) = \max(X_{i,j}(A), X_{i,k}(B) \cdot X_{i+k,j-k}(C) \cdot p(r))$
            end
        end
    end
end
return $X_{1,n}(S)$
```

The Penn Treebank is still frequently used in NLP research despite its age (> 20 years) and the disadvantage that it is not freely available. A sample of around 4000 trees is available with the python NLTK tool [LB02]. The TüBa-D/Z treebank on the other hand is still actively maintained and extended, free to use for academic purposes, and constitutes one of the largest parsed corpora available.

Example 8.2. *The following is an example parse tree from the Penn Treebank:*

```
(S
  (NP-SBJ
    (NP (NNP Pierre) (NNP Vinken))
    (, ,)
    (ADJP (NP (CD 61) (NNS years)) (JJ old))
    (, ,))
  (VP
    (MD will)
    (VP
    (VB join)
    (NP (DT the) (NN board))
    (PP-CLR (IN as) (NP (DT a) (JJ nonexecutive) (NN director)))
    (NP-TMP (NNP Nov.) (CD 29))))
  (. .))
```

The nonterminals that occur right before terminal words are called *part-of-speech (POS) tags*, e.g. "Dt" (determiner), "NN" (proper noun), or "JJ" (adjective).

The inner nonterminals of a parse tree, such as "NP" (noun phrase) or "PP" (prepositional phrase) are called *constituents* and represent categories that give a syntactic structure to the sentence.

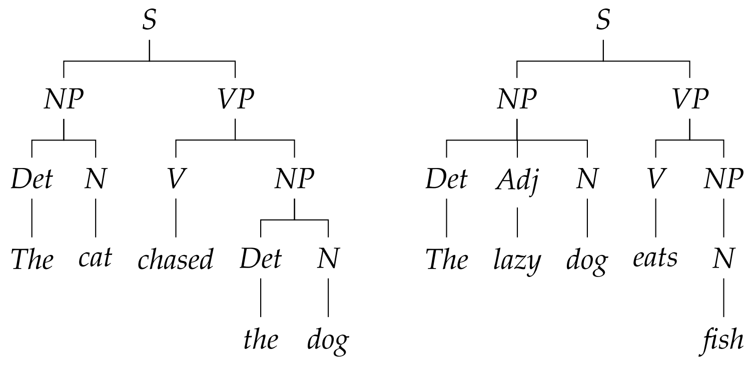
Training a PCFG from a Treebank Given a treebank, we can build a CFG by collecting all rules that appear in the trees. To obtain a PCFG we estimate the rule probabilities by counting: Let $c(A \to \alpha)$ denote the number of times a rule $A \to \alpha$ appears in the treebank and analogously let $c(A)$ be the number of times that the nonterminal A appears. Then the maximum-likelihood estimator for the probability of a rule $p(A \to \alpha)$ is given by the relative frequency of that rule, i.e.

$$\hat{p}(A \to \alpha) = \frac{c(A \to \alpha)}{c(A)}.$$

Chi and Geman show that PCFGs induced in this fashion are always consistent [CG98].

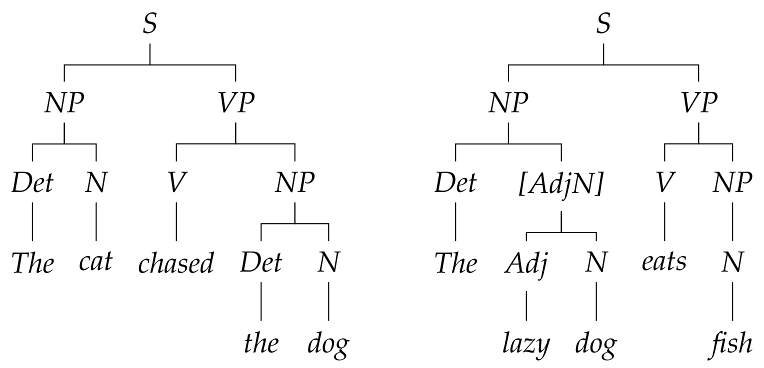
Note that the parse trees in a treebank are not necessarily binary trees and hence, the resulting PCFG would not be in CNF. Instead of transforming the PCFG into CNF (which is not always possible [1]), we first binarize the trees in the treebank before deriving the grammar.

Example 8.3. *Suppose we have a (very) small treebank consisting of the two parse trees below:*



The first tree is already binary so we only binarize the second tree by introducing a new nonterminal "[AdjN]":

[1]The transformation into CNF still yields an $\mathbb{R}_{>0}$-weighted grammar but it can happen that the weights do not define a probability distribution over the rules involving some nonterminal.



We finally obtain the following PCFG in CNF, estimating the rule probabilities by relative frequencies:

$$S \xrightarrow{1} NP\ VP \qquad Det \xrightarrow{1} the$$
$$NP \xrightarrow{0.5} Det\ N \qquad V \xrightarrow{0.5} chased$$
$$NP \xrightarrow{0.25} Det\ [AdjN] \qquad V \xrightarrow{0.5} eats$$
$$NP \xrightarrow{0.25} N \qquad [AdjN] \xrightarrow{1} Adj\ N$$
$$N \xrightarrow{0.5} dog \qquad Adj \xrightarrow{1} lazy$$
$$N \xrightarrow{0.25} cat$$
$$N \xrightarrow{0.25} fish$$

8.2.4 Refining PCFGs

Naively inducing a PCFG from a treebank does are not yield a good parsing model. One problem is that the nonterminal categories are too coarse, e.g. a nounphrase (NP) which is a subject NP is much more likely to expand to a pronoun than an object NP [KM03]. Hence it can be beneficial to distinguish between subject and object NPs. A general method to correct the "context-freeness" of the PCFG model is to re-introduce context to PCFGs by splitting nonterminals into subcategories.

One popular method proposed by Johnson [Joh98] is parent annotation. There every nonterminal in the training treebank is annotated with its parent nonterminal. Thus every nonterminal is split into at most $|\mathcal{X}|$ nonterminals, although in practice the size of the grammar grows much less. There are many other annotations that are usually applied to derive practical parsers (e.g. marking temporal or possessive nounphrases), see [KM03]. Petrov et al. propose to learn subcategories for nonterminals automatically [PBTK06]. In their approach, nonterminals in the grammar are repeatedly split and the rule probabilities of the PCFG are re-estimated using the Inside-Outside-algorithm (an EM-algorithm for unsupervised training of PCFGs). If splits lead to better parsing performance (on separate validation set) they are kept otherwise discarded.

8.2.5 Evaluating Parsers

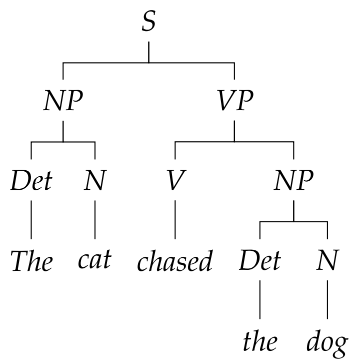
To evaluate statistical parsing methods, one usually applies the standard procedure used in statistics, data mining, or machine learning: The treebank is randomly partitioned into two (disjoint!) parts, a (large) *training set* (e.g. 90% of the data) and a smaller *test set* (e.g. 10% of the data). The training set is used to estimate the statistical model, and the test set is used for evaluation. We usually repeat this process several times to avoid any potential bias induced by the selection of the test set.

For the evaluation we run the parser on the test sentences and compare the parse tree produced by the parser to the "correct" parse tree found in the treebank (referred to as the "gold tree"). To compare two parse trees, several metrics are commonly used. We recall them briefly in the following.

PARSEVAL Metrics: Precision, Recall, F_1 The PARSEVAL metrics were proposed in [AFG$^+$91] to evaluate statistical parsers. They are still the most popular way to evaluate parsing accuracy.

For a nonterminal X and natural numbers $i \leqslant j$ we call $X(i,j)$ a *bracket* of a parse tree if the nonterminal X is the root of the subtree spanning the sub-sentence $s_i \dots s_j$.

Example 8.4. *Consider the sentence from before*



The set of all brackets of this sentence is given by

$$\{S(1,5), NP(1,2), Det(1,1), N(2,2), VP(3,5), V(3,3), NP(4,5), Det(4,4), N(5,5)\}$$

The *precision* of a parse tree t_{guess} produced by a parser w.r.t. the "correct" parse tree t_{gold} is defined as the number of correct brackets divided by the total number of brackets produced by the parser. More precisely, let B_{guess} denote the set of brackets appearing in the tree t_{guess} and analogously define B_{gold}, then

$$\text{Prec}(t_{\text{guess}}, t_{\text{gold}}) := \frac{|B_{\text{guess}} \cap B_{\text{gold}}|}{|B_{\text{guess}}|}$$

Note that a parser could achieve precision of 100% on a sentence $s_1 \cdots s_n$ by simply outputting a tree with the single bracket $S(1,n)$. Hence, precision should always be reported together with *recall*, which measures the proportion of correct brackets w.r.t. the total number of brackets in the gold tree:

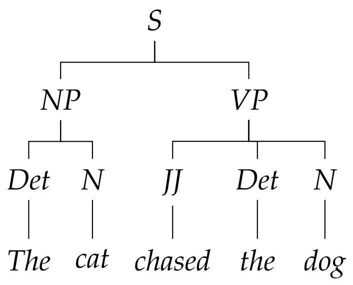
$$\text{Rec}(t_{\text{guess}}, t_{\text{gold}}) := \frac{|B_{\text{guess}} \cap B_{\text{gold}}|}{|B_{\text{gold}}|}$$

A nearly perfect recall could be produced by a parser that simply outputs a tree comprising lots of brackets.

Precision and recall are often condensed into a single number, the F-measure or F_1-score which is the harmonic mean of precision and recall:

$$F_1 := 2 \cdot \frac{\text{Prec} \cdot \text{Rec}}{\text{Prec} + \text{Rec}}$$

Example 8.5. *Consider a parser that outputs the parse tree below for the sentence "The cat chased the dog". For some reason, the parser produces the wrong POS tag for "chased" and decides to parse the last three words as a VP with three children.*



The set of brackets for this tree is given by

$$\{S(1,5), NP(1,2), Det(1,1), N(2,2), VP(3,5), JJ(3,3), Det(4,4), N(5,5)\}$$

Hence, the precision, recall and F_1 measure on this sentence are

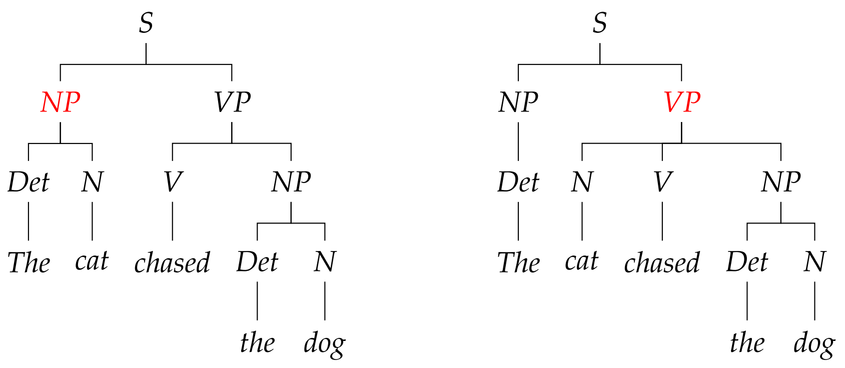
$$\text{Prec} = \frac{6}{8} = 0.75$$
$$\text{Rec} = \frac{6}{9} = 0.6\overline{6}$$
$$F_1 = \frac{12}{17} \approx 0.7059$$

Note that the F_1-score "punishes" large differences between precision and recall.

To evaluate the parser on a set of trees, the precision, recall, and F_1-score are averaged over all trees.

Crossing Brackets A *crossing bracket* is an error produced by a parser witnessed by a bracket $X(i, j)$ in the output tree and a bracket $Y(k, l)$ in the gold tree such that neither span contains the other, i.e. $[i, j] \not\subseteq [k, l] \wedge [k, l] \not\subseteq [i, j]$. Intuitively, this means that the parser does not even detect the rough syntactic structure and assigns a part of the sentence to the wrong subtree. According to [MS99] a high number of crossing brackets is considered "particularly dire".

Example 8.6. *Consider again the gold tree on the left and an output tree on the right.*



Here the bracket $VP(2, 5)$ in the second tree crosses the bracket $NP(1, 2)$ from the first tree.

Leaf-Ancestor Metric The leaf-ancestor (LA) metric was proposed by [Sam00] who argues that it better describes the informal notion of a "good" parse than the above PARSEVAL measures. This is especially relevant for comparing parsing performance over different treebanks as shown by [RVG07a, RvG07b].

The LA metric is computed for each leaf (i.e. terminal word) of a parse tree: Given a parse tree for a sentence and a particular leaf, we consider the path from the leaf to the root (the "lineage") as a word over the nonterminals $\mathcal{X}^*$. [2]

For a particular word in the sentence, the gold and guess tree induce the lineages l_1 and l_2. The LA score for this word is calculated using the Levenshtein distance (also called the edit distance) d_E between the two lineages l_1 and l_2:

$$LA = 1 - \frac{d_E(l_1, l_2)}{|l_1| + |l_2|}.$$

Note that the LA score is defined for a single word of a sentence. To obtain a score for a set of sentences there are two possibilities for averaging the scores:

- Compute the average LA score of each sentence and then take the average of these over the whole set. This number is called the *sentence-average* score.

[2]More precisely, [Sam00] considers trees encoded as parenthesized words and lineages are represented as words over $(\mathcal{X} \cup \{[,]\})^*$ in a more complicated way such that the set of lineages determines a tree uniquely.

- Average the LA score for each word over the whole set and then average these over all words. This number is called the *corpus-average* score.

8.3 Structural Annotations for Improved Parsing

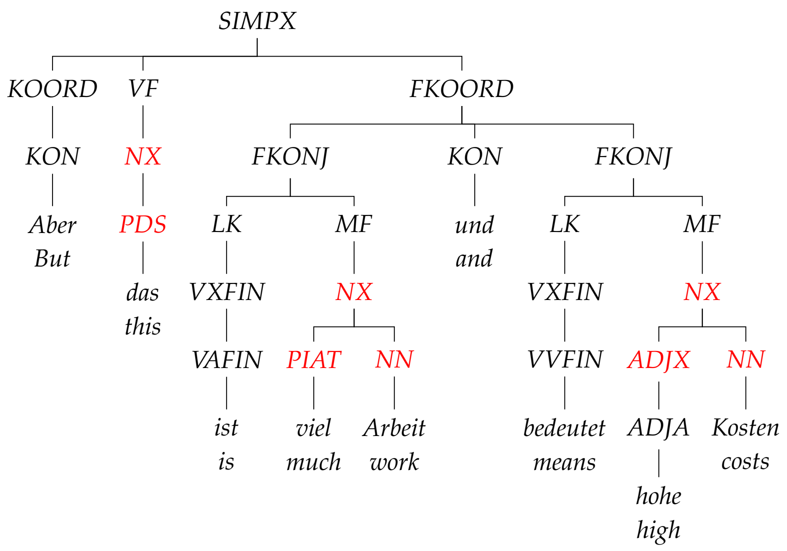
Previously proposed methods to improve the parsing accuracy of PCFGs mainly have focused on linguistically motivated annotations of the training trees.

Here, we propose a new annotation method which only depends on the graph-theoretic structure of the parse tree: We annotate each node by the dimension of the subtree rooted at it. Since it could be argued that any such graph-theoretic parameter can be used as annotation, we also consider annotating nodes by their height which is arguably the simplest measure of the structural complexity of trees.

We experimentally evaluate the impact of different annotation methods on parsing performance using the freely available TüBa-D/Z treebank of written German. More specifically, we study the dimension annotation (DA), height annotation (HA), and parent annotation (PA) and their combinations (DA+PA and HA+PA).

Our results in Section 8.3.2 show that the dimension annotation substantially improves both parsing accuracy and parsing speed. Accuracy can be further increased by combining dimension and parent annotation.

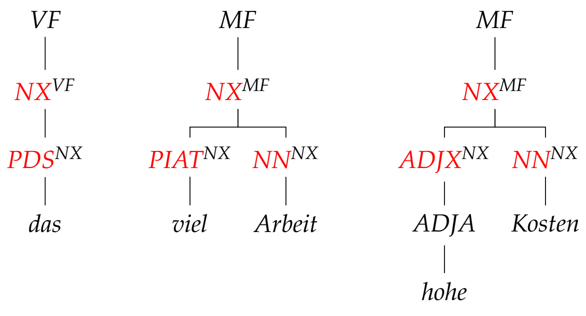
Example 8.7. *We illustrate the different annotation methods using the following example from the TüBa-D/Z treebank (where we have provided a literal translation below each word):*



The parts of the tree involving the nonterminal "NX" are highlighted in red. As a result we obtain the following rules for "NX" if we derive a PCFG from this single tree:

$$\begin{aligned} NX &\xrightarrow{\frac{1}{3}} PDS \\ NX &\xrightarrow{\frac{1}{3}} PIAT \quad NN \\ NX &\xrightarrow{\frac{1}{3}} ADJX \quad NN \end{aligned}$$

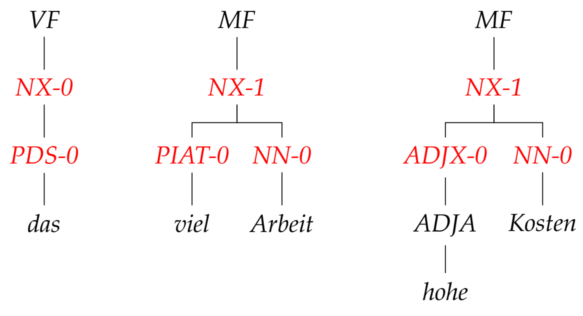
With parent annotation, the respective parts of the tree look as follows



Hence, we extract a different set of rules from this tree:

$$\begin{aligned} NX^{VF} &\xrightarrow{1} PDS^{NX} \\ NX^{MF} &\xrightarrow{0.5} PIAT^{NX} \quad NN^{NX} \\ NX^{MF} &\xrightarrow{0.5} ADJX^{NX} \quad NN^{NX} \end{aligned}$$

Dimension annotation attaches to each nonterminal the dimension D *of the subtree rooted at it by appending the string "-D" to its name (note that we could skip the annotation for POS tags which is always "-0"):*



Dimension annotation leads to the following set of rules:

$$
\begin{aligned}
\mathit{NX\text{-}0} &\xrightarrow{1} \mathit{PDS\text{-}0} \\
\mathit{NX\text{-}1} &\xrightarrow{0.5} \mathit{PIAT\text{-}0} \quad \mathit{NN\text{-}0} \\
\mathit{NX\text{-}1} &\xrightarrow{0.5} \mathit{ADJX\text{-}0} \quad \mathit{NN\text{-}0}
\end{aligned}
$$

Annotating the training trees with their dimension is similar to unfolding the (unweighted) CFG derived from the treebank w.r.t. dimension and then estimating the rule probabilities by relative frequencies. However, this unfolding approach does not provide a direct correspondence between the nonterminals in the unfolding and the nonterminals in the training data, which complicates training. Hence, annotating training trees is simpler and also allows us to use an existing parser as a black box and feeding it the annotated training data.

8.3.1 Experimental Setup and Methods

In this section we briefly describe our experimental procedures and setup.

Data and Tools As dataset we use the TüBa-D/Z treebank of written German [THK+03] version 8.0, comprising 75,408 parse trees of sentences from the German newspaper "taz".

For the annotation of the training trees (with dimension or height) we use python NLTK [LB02]. We use the Stanford parser [KM03] to train a PCFG and parse the test data. Since the dimension of a tree is not invariant w.r.t. binarizations (see Section 3.6.3), we *first* annotate nodes with their dimension before using the parser to binarize the trees and derive a PCFG. We use the metrics described in Section 8.2.5 which are implemented in the Stanford parser to evaluate the parser on the test sentences.

It is important to note that we do not feed raw sentences to the parser but rather tag words with their "correct" POS tags from the treebank. Hence we start the parse of each sentence with the pre-terminals (i.e. the nodes of height 1). While it is slightly unrealistic to assume perfect POS tags this is a common assumption and makes our work comparable to others like [RM08]. Furthermore, this avoids any parse failures due to unknown words. In practice, one would smooth the PCFG to account for unknown words (i.e. set aside a small probability for unseen rules).

We ran our experiments on a machine with an Intel i7 2.7 GHz CPU and 8 GB of memory. The experiments took about one week to complete (using a single core since memory turned out to be the main bottleneck). All our scripts and output data can be obtained from `https://github.com/mschlund/nlp-newton`.

Randomization We sample our training- and test-data 10 times randomly from the treebank for each of the six annotation methods (none, PA, HA, DA, HA+PA, DA+PA). This enables us to study the relation between accuracy and the amount of training data available and to assess the variance of the performance of the PCFG (cf. Figure 8.2).

For each sample size N from $\{5k, 10k, 20k, \ldots, 70k\}$ we draw a random sample of size N from the set of all 75408 trees in the treebank. The first 90% of this sample was used as training set and the remaining 10% as test set. We then evaluated each of our annotation methods on this same training/test set. The whole process was repeated ten times each, yielding 480 experiments altogether. For each experiment we evaluated parsing performance according to the evaluation measures described before, as well as the parsing speed and the size of the derived grammar. Each of these numbers is then averaged over the 10 independent random trials. To ensure perfect reproducibility we recorded the seeds that we used to seed the random generator.

8.3.2 Experimental Results

Our main results are collected in Table 8.2.

Parsing Accuracy As a baseline we use the un-annotated PCFG which gives an F_1-score of 84.8% in our experiments. Rafferty and Manning [RM08] report an F_1-score of 88% on a previous release of the TüBa-D/Z treebank (comprising only $20k$ sentences of length at most 40). However, the absolute improvements in F_1 we observe using parent annotation are consistent with their work, e.g. our experiments show an absolute increase of 3.4% with parent annotation while [RM08] report a 3.1% increase. We suspect the different datasets are the main reason for this difference, especially since their data only contained sentences of length at most 40 which are easier to parse. Considering only sentences up to length 40, our experiments yield scores that are about 1% higher (e.g. $91.2F_1$ for the DA+PA annotation).

The DA, PA, and HA annotation methods alone lead to comparable improvements w.r.t. constituency measures with small advantages for the two structural annotations (DA,HA). Considering the LA metric shows that HA and DA have a clear advantage of 3% over PA.

Both structural annoation methods can be fruitfully combined with parent annotation, leading to further improvements in the metrics. However, the HA+PA combination seems to suffer from the large blowup in grammar size and the resulting data-sparseness problem: Considering the learning curves in Figure 8.2 we see that the HA+PA method needs a lot of training data to reach acceptable performance and that the performance of the HA+PA combination does not reach a plateau yet at a sample size of $70k$.

Altogether, the DA+PA combination is the most precise one w.r.t. all metrics. It provides absolute increases of 5.6% F_1 and 7.4–8.4% LA-score and offers a relative reduction of crossing brackets by 27%. Most importantly, the DA+PA method yields a relative increase of 60% in the number of exactly parsed sentences compared to the PCFG baseline.

Parsing Speed The most surprising result of our experiments is that the dimension annotation significantly *increases* the parsing speed despite the growth of the induced grammar. For common parsing algorithms (such as CYK or Earley), parsing speed is inversely proportional to the size of the grammar G. The decrease in parsing speed for parent annotation follows this hypothesis almost exactly: $|G|$ grows by a factor of 1.6 and parsing speed drops by a factor of roughly $1/(1.6)$.

However, despite the fact that DA and HA increase the size of the grammar by a factor of 2.4 and 3.6, respectively, parsing speed under these annotations also *increases* by a factor of almost 3.5 and 1.8, respectively.

A possible explanation for this phenomenon is the fact that (for a grammar in CNF) a nonterminal of dimension d can only be produced either by combining one of dimension d with one of dimension strictly less than d or by two of dimension exactly $d - 1$. Since the dimensions involved are typically very small this restricts the search space significantly.

			PARSEVAL		Leaf-Ancestor		Crossing brackets	
Annotation	$\|G\|$	Speed $\pm$ stderr	F_1	exact	LA (s)	LA (c)	# CB	zero CB
None	21009	1.74 ± 0.04	84.8	24.4	84.0	79.7	1.17	58.5
PA	34192	1.07 ± 0.01	88.2	31.8	86.6	82.9	1.07	61.8
HA	76096	3.06 ± 0.03	88.7	33.7	89.8	86.2	0.93	65.2
HA+PA	130827	2.20 ± 0.04	89.2	36.8	90.8	87.0	0.95	65.4
DA	49798	$\mathbf{6.02} \pm 0.10$	88.5	31.8	89.7	86.1	0.90	64.9
DA+PA	84947	4.04 ± 0.07	**90.4**	**39.1**	**91.4**	**88.1**	**0.85**	**67.2**

Table 8.2: Average grammar size, parsing speed, and parsing accuracies according to various metrics for random samples of 70k trees (split into 63k training and 7k test trees) for six different annotation methods, including parent annotation (PA), height annotation (HA), dimension annotation (DA), and their respective combinations. All numbers are averaged over 10 independent runs. $|G|$ denotes the number of rules in the grammar, parsing speed is measured in sentences per second. LA scores are reported as sentence-level (s) and corpus-level (c) averages, respectively. All accuracies reported in % (except # CB – the average number of crossing brackets per sentence).

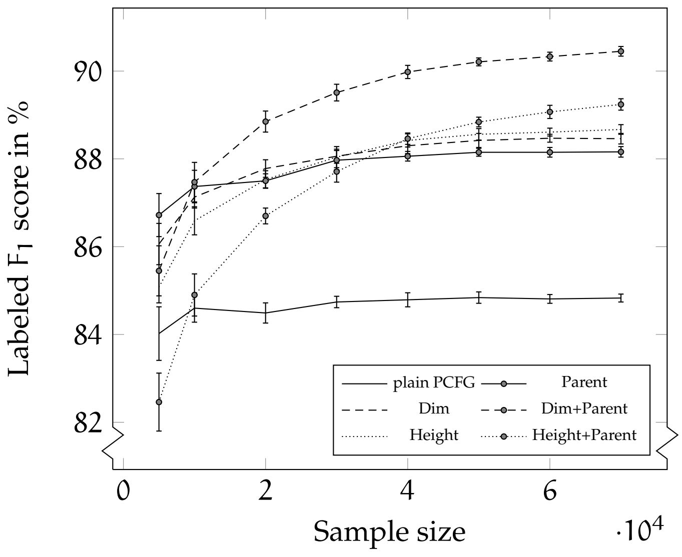


Figure 8.2: Learning curves for different annotation methods. Average F_1 with standard deviation for random samples of various sizes (10 independent runs each).

8.4 Conclusions, Related, and Future Work

In this chapter we have demonstrated that tree dimension has interesting applications in the field of computational linguistics and natural language processing. In particular, we have given evidence that tree dimension is a plausible (albeit very rough) measure of syntactic complexity of natural language. Furthermore, we have demonstrated that statistical parsing can benefit from considering dimension information of parse trees.

The experiments reported here have only scratched the surface and suggest many promising venues for future work:

- Investigate the combination of dimension with more features, also for other probabilistic parsing models (e.g. CRFs instead of PCFGs).
- Experiment with more languages and parsing methods (like Earley or left-corner parsing) to pinpoint the exact reason why dimension annotation is beneficial. In the long run it would be desirable to have a thorough theoretic understanding of why e.g. the parsing speed improves by dimension annotation.
- Study whether complexity features (such as dimension or pathwidth) can also improve *dependency parsing* – see below.
- Provide a thorough theoretical analysis of the speedup effect for parsing when using dimension annotation. Note that in principle, this effect might be confined to

CYK-style parsers or be specific to the German treebank we considered. (although preliminary experimentation suggests that the speedup effect does not depend on the treebank).

Dependency Grammar, Parsing, and Pathwidth In the framework of *dependency grammar*, introduced by Tesnière [Tes53], the set T of syntactic analyses is the set of directed graphs with words as nodes and edges encoding syntactic dependencies between words. A classical formal treatment of dependency grammar was developed by Hays [Hay64] and Gaifman [Gai65]. More recent work in statistical NLP usually does not use such descriptive formal models but approaches the parsing problem by training probabilistic models (either generative or discriminative) from a large set of human-generated dependency structures. For a comprehensive introduction to dependency parsing we recommend the report by Nivre [Niv05].

Kornai and Tuza [KT92] propose a memory-restricted machine model for processing sentence structures given as dependency graphs. Their model has access to a small set ("at most 6 or 7") of memory cells that may hold vertices of the dependency graph. A dependency graph is processed by moving its vertices from the input graph to an output via the internal memory. The restriction is that a vertex can only be output if all its dependents have been output. Kornai and Tuza introduce the *narrowness* of a graph as the minimum amount of memory required to process it in this fashion and prove that the narrowness of a graph is exactly its pathwidth plus one. This establishes pathwidth as a natural measure for the complexity of dependency structures. Our theoretical results from Section 3.6.2 which connect tree dimension and pathwidth, provide an important link to this work and suggest that dependency parsing can be improved as well by considering structural complexity features like tree dimension.

9 Conclusions

Here we briefly review our results and outline some ideas for future work.

9.1 Contributions

9.1.1 Theoretical Foundations (Chapter 3)

We have described Newton's method as an unfolding of context-free grammars w.r.t. dimension (note that this idea already appeared in [EL11]). From the unfolding we have extracted the definition of Newton's method via derivatives which is more suitable for an algorithmic treatment than the unfolding. In particular, the unfolding scheme gives an effective definition of the value δ_d which represents the difference $F(\nu_d) - \nu_d$ over semirings. In [EKL10] it was shown that such a δ_d always exists and that the sequence of Newton approximations is independent of the choice of δ_d.

Exploiting the combinatorial properties of tree dimension, we have proved a general theorem on the convergence speed of Newton's method over commutative semirings (Theorem 3.33). Our result implies that Newton's method computes the solution of algebraic systems over commutative k-collapsed semirings in $\mathcal{O}(n + \log\log k)$ steps.

A theoretical application of the convergence result that we have not mentioned so far is the generalization of Parikh's theorem to bounded multiplicities shown in [LS13, LS15]. We show that modulo $k = k + 1$ we can transform the rational expressions computed by Newton's method into "weighted" semilinear expressions (cf. Chapter 5), i.e. finite sums of "weighted" linear expressions of the form $k' \cdot cP^*$ with $k' \in [k]$. Note that Petre [Pet99] showed that Parikh's theorem does not hold for multiplicities in $\mathbb{N}$.

More specifically he defined the hierarchy of "semilinear", "rational", and "algebraic" power series over $\mathbb{N}\langle\langle\Sigma^{\oplus}\rangle\rangle$ and proved its strictness. Our results in [LS13, LS15] complement this work and show that the hierarchy collapses over $\mathbb{N}_k\langle\langle\Sigma^{\oplus}\rangle\rangle$.

The convergence theorem is also central to our application for Datalog provenance (cf. Chapter 6) since it allows us to efficiently compute regular expressions representing provenance over the general semiring $\mathbb{N}_k\langle\langle\Sigma^{\oplus}\rangle\rangle$.

Regarding the combinatorial theory of tree dimension, we have proved a tight relation between the dimension (aka. Strahler number) and the pathwidth of trees in Section 3.6.2. This settles an open question of David Eppstein[1].

9.1.2 Algorithms and Implementation (Chapters 4 and 5)

We have described generic algorithms and data structures for Newton's method, both for general semirings and for the special semiring of semilinear sets. Our implementation FPSOLVE provides a generic and efficient C++ library which is easy to extend. Currently we are the primary user of FPSOLVE which was helpful for experimenting with Newton's method. We hope that we can continue to extend and maintain FPSOLVE.

9.1.3 Applications

A large part of this thesis is concerned with the applications of the theoretical results in various areas.

Datalog Provenance (Chapter 6) Using our results on Newton's method we have shown that the general k-collapsed semirings $\mathbb{N}_k\langle\langle\Sigma^{\oplus}\rangle\rangle$ can be used to represent the provenance of recursive Datalog programs. Our representation via shared regular expression is of polynomial size (in the EDB) and is efficiently computable via Newton's method. We have described how to specialize these expressions to the $\mathsf{Why}(\Sigma)$ semiring with a limited increase in size (at most by a factor of $\mathcal{O}(\log|\Sigma|)$). This shows that a special treatment of semirings such as $\mathsf{Why}(\Sigma)$ as proposed in [DMRT14] is not necessary.

Sub-/Superword Closure of CFGs (Chapter 7) We have answered two open questions of Gruber, Holzer, and Kutrib [GHK09]: Our work improves the upper bound on the size of an NFA recognizing the subword closure from $2^{2^{\mathcal{O}(|G|)}}$ to $2^{\mathcal{O}(|G|)}$ which is asymptotically "tight". Note however, that there is still some gap between our upper bound of $\mathcal{O}(3^{|G|})$ for grammars in C2F and the straightforward lower bound of $\Omega(2^{|G|})$. Furthermore, we have shown that the trivial upper bound of $2^{2^{\mathcal{O}(|G|)}}$ for the size of the DFA representing the sub- or superword closure of a CFG is essentially tight. A simple but surprising result is that equivalence of NFAs modulo sub-/superword closure is (only!) coNP-complete (note that general NFA equivalence is PSPACE-complete). The result

[1]See `https://en.wikipedia.org/wiki/Talk:Pathwidth` for a discussion.

is surprising since the seemingly related problem of NFA equivalence modulo prefix, suffix, or factor closure stays PSPACE-complete [RSX12].

Our work on the subword closure of CFGs has already been used and extended by Karandikar, Niewerth, and Schnoebelen who studied the descriptional and computational complexity of subword interiors of regular languages in [KNS14, KS14].

Natural Language Processing (Chapter 8) Motivated by the combinatorial properties of tree dimension, we have suggested that dimension is a rough but useful measure of syntactic complexity. We have successfully used tree dimension as annotation to improve probabilistic parsing (evaluated for the TüBa-D/Z treebank using the Stanford parser). Besides improving the accuracy of a PCFG trained on the annotated treebank the dimension annotation significantly speeds up parsing in our experiments. Our annotation heuristic was implemented as a feature in the current version of the Stanford parser by Christopher Manning [2].

9.2 Open Problems and Ideas for Future Work

Theory An interesting theoretical problem is to investigate the distribution of the dimension for CFGs with multiple nonterminals. For the derivation trees of the grammar $X \to aXX \mid c$ (the proper binary trees), Flajolet, Raoult, and Vuillemin [FRV79] give a closed form for the number of trees with n leaves and dimension less than d, i.e. for $\mathsf{camb}_{G^{<d},X}(a^{n-1}c^n)$. They show that the expected dimension of a random binary tree with n leaves is $1/2 \log_2 n$ and that the distribution is tightly concentrated around this mean. This result implies a much faster convergence of Newton's method for this special grammar. It would be highly interesting to derive a similar result for arbitrary context-free grammars.

Language-Based Testing We would like to extend the grammar testing tool we described in Section 5.4 to incorporate more semirings. Specifically, it could be useful to study more specialized commutative semirings for which the semiring operations are cheaper to compute than for semilinear sets (e.g. the tropical semiring as shown in Example 5.17). We also want to apply the method using non-commutative semirings for which we can effectively compute Newton's method, like the matrix semirings over a commutative semiring as described in Section 3.4.4.

Especially the matrix semirings over a commutative semiring could be useful for a refinement strategy: If $h(L_1) = h(L_2)$ for a homomorphism $h : \Sigma^* \to S$ into a commutative semiring S we would try to find a refinement that distinguishes the languages, e.g. an interpretation $\iota' \to S^{[2]\times[2]}$ such that $h'(L_1) \neq h'(L_2)$. The intuition behind this is that

[2]See `https://github.com/stanfordnlp/CoreNLP/blob/master/src/edu/stanford/nlp/parser/lexparser/TrainOptions.java`

since matrix semirings are non-commutative they can distinguish the ordering of letters in Σ to some degree (depending on the dimensionality of the matrix).

To make this more precise, we pose the following question (we have not yet experimented with it sufficiently to call it a conjecture):

Question 9.1. *Is the following statement true?*

For any two CFGs G_1, G_2 *with* $\mathcal{L}(G_1) \neq \mathcal{L}(G_2)$ *there exists a* $k \in \mathbb{N}$ *and an interpretation* $\iota : \Sigma \to (\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle)^{[k]\times[k]}$ *into the semiring of* $k \times k$ *matrices over* $\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle$ *such that* $s_1 \neq s_2$ *for the least solutions* $s_1, s_2 \in (\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle)^{[k]\times[k]}$ *of the associated algebraic systems over* $(\mathbb{B}\langle\langle\Sigma^{\oplus}\rangle\rangle)^{[k]\times[k]}$.

Hence our question is, if the semiring of $k\times k$-matrices over the semilinear sets is powerful enough to distinguish two in-equivalent context-free languages (note that k may depend on the grammars). A related morphism that could prove useful to distinguish languages is the generalization of the Parikh-image to matrices introduced by Mateescu, Salomaa, Salomaa, and Yu in [MSSY01].

Our current work only discussed methods to show the in-equivalence of grammars. Although equivalence of CFGs is not even semi-decidable, it would be useful in practice to explore (necessarily incomplete) methods that can certify the equivalence of CFGs, e.g. by exhibiting a suitable bisimulation.

Datalog We have considered standard Datalog without negation or aggregation operations. A logical next step is to extend our provenance representation to (potentially recursive) queries with aggregation. Amsterdamer et al. [ADT11b] claim that the semiring framework is insufficient to capture provenance even for non-recursive queries and suggest semi-modules for representing provenance. Hence, we want to investigate their framework in greater detail and possibly adapt our provenance representation to semi-modules.

Provenance has recently been used by Deutch et al. [DMT14] for the analysis of so called data-dependent-processes (DDPs). According to their framework DDPs are finite automata with some optional transitions whose existence depends on the results of a database query. Deutch et al. consider finite state DDPs, specifications over finite runs, and provenance for non-recursive queries (albeit with aggregation). We believe that our results on provenance and Newton's method can be used to generalize the approach of Deutch et al. in several directions, i.e. we want to (1) consider recursive Datalog queries, (2) specification over infinite runs, and (3) infinite state DDPs, e.g. given as pushdown automata. Note that in the model investigated by Deutch et al., the "data-dependency-analysis" can be run as a preprocessing step and does not add expressive power to the automata model.

Bibliography

[ABLM08] Carlos Ansótegui, María Luisa Bonet, Jordi Levy, and Felip Manyà. Measuring the Hardness of SAT Instances. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 1*, AAAI'08, pages 222–228. AAAI Press, 2008.

[ABT08] Mohamed Faouzi Atig, Ahmed Bouajjani, and Tayssir Touili. On the Reachability Analysis of Acyclic Networks of Pushdown Systems. In *CONCUR 2008*, pages 356–371, 2008.

[ACBJ04] Parosh Aziz Abdulla, Aurore Collomb-Annichini, Ahmed Bouajjani, and Bengt Jonsson. Using Forward Reachability Analysis for Verification of Lossy Channel Systems. *Formal Methods in System Design*, 25(1):39–65, 2004.

[ADT11a] Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for Aggregate Queries. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, pages 153–164, 2011.

[ADT11b] Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for Aggregate Queries. In *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '11, pages 153–164, New York, NY, USA, 2011. ACM.

[AFG$^+$91] S. Abney, S. Flickenger, C. Gdaniec, C. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. Klavans, M. Liberman, M. Marcus, S. Roukos, B. Santorini, and T. Strzalkowski. Procedure for Quantitatively Comparing the Syntactic Coverage of English Grammars. In E. Black, editor, *Proceedings of the Workshop on Speech and Natural Language*, HLT '91, pages 306–311, Stroudsburg, PA, USA, 1991. Association for Computational Linguistics.

[AHL08] Roland Axelsson, Keijo Heljanko, and Martin Lange. Analyzing Context-Free Grammars Using an Incremental SAT Solver. In *ICALP (2)*, pages 410–422, 2008.

[Aho68] Alfred V. Aho. Indexed Grammars – An Extension of Context-Free Grammars. *J. ACM*, 15(4):647–671, October 1968.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[And98] H. R. Andersen. An Introduction to Binary Decision Diagrams. Technical report, IT University of Copenhagen, April 1998. Lecture notes.

[ASV13] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. How to Combine Widening and Narrowing for Non-Monotonic Systems of Equations. In Hans-Juergen Boehm and Cormac Flanagan, editors, *PLDI*, pages 377–386. ACM, 2013.

[BC96] Alexandre Boudet and Hubert Comon. Diophantine Equations, Presburger Arithmetic and Finite Automata. In *Proceedings of CAAP '96*, volume 1059 of *LNCS*, pages 30–43. Springer, 1996.

[BÉ09] Stephen L. Bloom and Zoltán Ésik. Axiomatizing rational power series over natural numbers. *Inf. Comput.*, 207(7):793–811, 2009.

[Ben77] David B. Benson. Some Preservation Properties of Normal Form Grammars. *SIAM Journal on Computing*, 6(2):381–402, 1977.

[Ber09] Dennis S. Bernstein. *Matrix Mathematics*. Princeton University Press, 2009.

[BGB10] Aydın Buluç, John R. Gilbert, and Ceren Budak. Solving Path Problems on the GPU. *Parallel Comput.*, 36(5-6):241–253, 2010.

[BK14] Olaf Beyersdorff and Oliver Kullmann. Unified Characterisations of Resolution Hardness Measures. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pages 170–187. Springer International Publishing, 2014.

[BKWC01] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and Where: A Characterization of Data Provenance. In *Database Theory — ICDT 2001*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330. Springer Berlin Heidelberg, 2001.

[BLS15] Georg Bachmeier, Michael Luttenberger, and Maximilian Schlund. Finite Automata for the Sub- and Superword Closure of CFLs: Descriptional and Computational Complexity. In *Language and Automata Theory and Applications - 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings*, pages 473–485, 2015.

[Boz99] S. Bozapalidis. Equational Elements in Additive Algebras. *Theory Comput. Syst.*, 32(1):1–33, 1999.

[Bru14] Matthias Brugger. Analyse von kontextfreien Grammatiken mit Hilfe eines SMT-Solvers. Bachelor's thesis, TU München, 2014.

[Bry86] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *Computers, IEEE Transactions on*, C-35(8):677–691, Aug 1986.

[CCT09] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in Databases: Why, How, and Where. *Found. Trends databases*, 1(4):379–474, April 2009.

[CDF96] Kevin Cattell, Michael J. Dinneen, and Michael R. Fellows. A Simple Linear-Time Algorithm for Finding Path-Decompositions of Small Width. *Information Processing Letters*, 57(4):197 – 203, 1996.

[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.

[CG98] Zhiyi Chi and Stuart Geman. Estimation of Probabilistic Context-free Grammars. *Comput. Linguist.*, 24(2):299–305, June 1998.

[Cou91] B. Courcelle. On Constructing Obstruction Sets of Words. *EATCS Bulletin*, 44:178–185, 1991.

[Cro99] Matthew Crocker. Mechanisms for Sentence Processing. In *Language Processing*, pages 191–232. Psychology Press, London, UK, 1999.

[CS63] N. Chomsky and M.P. Schützenberger. *Computer Programming and Formal Systems*, chapter The Algebraic Theory of Context-Free Languages, pages 118 – 161. North Holland, 1963.

[CWW00] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Trans. Database Syst.*, 25(2):179–227, June 2000.

[Dic13] Leonard Eugene Dickson. Finiteness of the Odd Perfect and Primitive Abundant Numbers with n Distinct Prime Factors. *American Journal of Mathematics*, 35(4):pp. 413–422, 1913.

[DKV09] M. Droste, W. Kuich, and H. Vogler. *Handbook of Weighted Automata*. Springer, 2009.

[DMRT14] Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. Circuits for Datalog Provenance. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014.*, pages 201–212, 2014.

[DMT14] Daniel Deutch, Yuval Moskovitch, and Val Tannen. A Provenance Framework for Data-dependent Process Analysis. *Proc. VLDB Endow.*, 7(6):457–468, February 2014.

[DTR] Evan Driscoll, Aditya Thakur, and Thomas Reps. WALi: the Weighted Automata Library. `https://research.cs.wisc.edu/wpis/wpds/`.

[ÉK04] Zoltán Ésik and Werner Kuich. Inductive *-Semirings. *Theor. Comput. Sci.*, 324(1):3–33, 2004.

[EKL07a] J. Esparza, S. Kiefer, and M. Luttenberger. An Extension of Newton's Method to ω-Continuous Semirings. In *DLT*, pages 157–168, 2007.

[EKL07b] J. Esparza, S. Kiefer, and M. Luttenberger. On Fixed Point Equations over Commutative Semirings. In *STACS*, pages 296–307, 2007.

[EKL08] J. Esparza, S. Kiefer, and M. Luttenberger. Derivation Tree Analysis for Accelerated Fixed-Point Computation. In *DLT*, pages 301–313, 2008.

[EKL10] J. Esparza, S. Kiefer, and M. Luttenberger. Newtonian Program Analysis. *J. ACM*, 57(6):33, 2010.

[EKSW04] Keith Ellul, Bryan Krawetz, Jeffrey Shallit, and Ming-wei Wang. Regular Expressions: New Results and Open Problems. *J. Autom. Lang. Comb.*, 9(2-3):233–256, September 2004.

[EL11] Javier Esparza and Michael Luttenberger. Solving Fixed-Point Equations by Derivation Tree Analysis. In *CALCO*, pages 19–35, 2011.

[ELS14a] Javier Esparza, Michael Luttenberger, and Maximilian Schlund. A Brief History of Strahler Numbers. In *Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings*, pages 1–13, 2014.

[ELS14b] Javier Esparza, Michael Luttenberger, and Maximilian Schlund. FPsolve: A Generic Solver for Fixpoint Equations over Semirings. In *Implementation and Application of Automata - 19th International Conference, CIAA 2014, Giessen, Germany, July 30 - August 2, 2014. Proceedings*, pages 1–15, 2014.

[ELS15] Javier Esparza, Michael Luttenberger, and Maximilian Schlund. FPsolve: A Generic Solver for Fixpoint Equations over Semirings. *International Journal of Foundations of Computer Science*, 2015.

[Ers58] A. P. Ershov. On Programming of Arithmetic Operations. *Commun. ACM*, 1(8):3–9, 1958.

[ERV81] Andrzej Ehrenfeucht, Grzegorz Rozenberg, and Dirk Vermeir. On ET0L Systems with Finite Tree-Rank. *SIAM J. Comput.*, 10(1):40–58, 1981.

[Ési08] Z. Ésik. Iteration Semirings. In *Developments in Language Theory*, pages 1–20, 2008.

[EY05] Kousha Etessami and Mihalis Yannakakis. Recursive Markov Chains, Stochastic Grammars, and Monotone Systems of Nonlinear Equations. In *STACS 2005, 22nd Annual Symposium on Theoretical Aspects of Computer Science, Stuttgart, Germany, February 24-26, 2005, Proceedings*, pages 340–352, 2005.

[EY09] Kousha Etessami and Mihalis Yannakakis. Recursive Markov Chains, Stochastic Grammars, and Monotone Systems of Nonlinear Equations. *J. ACM*, 56(1), 2009.

[FBS12] Juliana Freire, Philippe Bonnet, and Dennis Shasha. Computational Reproducibility: State-of-the-art, Challenges, and Database Research Opportunities. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 593–596, New York, NY, USA, 2012. ACM.

[FKSS08] J. Freire, D. Koop, E. Santos, and C.T. Silva. Provenance for Computational Tasks: A Survey. *Computing in Science Engineering*, 10(3):11–21, May 2008.

[FL02] Alain Finkel and Jérôme Leroux. How to Compose Presburger-Accelerations: Applications to Broadcast Protocols. In *Proceedings of FST TCS 2002*, pages 145–156, 2002.

[FRV79] P. Flajolet, J.-C. Raoult, and J. Vuillemin. The Number of Registers Required for Evaluating Arithmetic Expressions. *Theor. Comput. Sci.*, 9:99–125, 1979.

[Gai65] Haim Gaifman. Dependency Systems and Phrase-Structure Systems. *Information and Control*, 8(3):304 – 337, 1965.

[GH08] Hermann Gruber and Markus Holzer. Finite Automata, Digraph Connectivity, and Regular Expression Size. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, pages 39–50, 2008.

[GH14] Hermann Gruber and Markus Holzer. From finite automata to regular expressions and back-a summary on descriptional complexity. In *Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014.*, pages 25–48, 2014.

[GHK09] Hermann Gruber, Markus Holzer, and Martin Kutrib. More on the Size of Higman-Haines Sets: Effective Constructions. *Fundam. Inf.*, 91(1):105–121, January 2009.

[GK14] Matthew Gwynne and Oliver Kullmann. A Framework for Good SAT Translations, with Applications to CNF Representations of XOR Constraints. *arXiv preprint arXiv:1406.7398*, 2014.

[GKP89] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics - A Foundation for Computer Science*. Addison-Wesley, 1989.

[GKT07] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance Semirings. In *PODS*, pages 31–40, 2007.

[Goo98] Joshua T. Goodman. *Parsing Inside-Out*. PhD thesis, Cambridge, MA, USA, 1998. AAI9832377.

[GR14] Carlos Gómez-Rodríguez. Finding the Smallest Binarization of a {CFG} is NP-Hard. *Journal of Computer and System Sciences*, 80(4):796 – 805, 2014.

[Gre09a] Todd J. Green. *Collaborative Data Sharing with Mappings and Provenance*. PhD thesis, University of Pennsylvania, 2009. Rubinoff Dissertation Award; honorable mention for Jim Gray Dissertation Award.

[Gre09b] Todd J. Green. Containment of Conjunctive Queries on Annotated Relations. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, pages 296–309, New York, NY, USA, 2009. ACM.

[GS68] S. Ginsburg and E. Spanier. Derivation-Bounded Languages. *Journal of Computer and System Sciences*, 2:228–250, 1968.

[GVL96] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[Har02] Theodore E. Harris. *The Theory of Branching Processes*. Courier Corporation, 2002.

[Hay64] David G. Hays. Dependency Theory: A Formalism and Some Observations. *Language*, pages 511–525, 1964.

[HGL11] Shan Shan Huang, Todd J. Green, and Boon Thau Loo. Datalog and Emerging Applications: An Interactive Tutorial. In *SIGMOD*, 2011.

[Hig52] Graham Higman. Ordering by Divisibility in Abstract Algebras. *Proceedings of the London Mathematical Society*, s3-2(1):326–336, 1952.

[HJJ$^+$95] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic Second-Order Logic in Practice. In *Proceedings of TACAS '95*, volume 1019 of *LNCS*, 1995.

[HMW10] Peter Habermehl, Roland Meyer, and Harro Wimmel. The Downward-Closure of Petri Net Languages. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and PaulG. Spirakis, editors, *Automata, Languages and Programming*, volume 6199 of *Lecture Notes in Computer Science*, pages 466–477. Springer Berlin Heidelberg, 2010.

[HN10] Géza Horváth and Benedek Nagy. Pumping Lemmas for Linear and Nonlinear Context-Free Languages. *CoRR*, abs/1012.0023, 2010.

[Hor45] R. E. Horton. Erosional development of streams and their drainage basins: hydro-physical approach to quantitative morphology. *Geological Society of America Bulletin*, 56(3):275–370, 1945.

[Hun73] H. B. Hunt, III. On the Time and Tape Complexity of Languages I. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 10–19, New York, NY, USA, 1973. ACM.

[Huy80] Thiet-Dung Huynh. The Complexity of Semilinear Sets. In *Automata, Languages and Programming*, volume 85 of *LNCS*, pages 324–337. Springer, 1980.

[Joh98] Mark Johnson. PCFG Models of Linguistic Tree Representations. *Computational Linguistics*, 24(4):613–632, 1998.

[Kem79] R. Kemp. The Average Number of Registers Needed to Evaluate a Binary Tree Optimally. *Acta Informatica*, 11:363–372, 1979.

[Ker70] Leslie R. Kerr. *The Effect of Algebraic Structure on the Computational Complexity of Matrix Multiplication*. PhD thesis, Cornell University, Ithaca, NY, USA, 1970.

[Ker14] Michael Kerscher. Symbolic Representations of Semilinear Sets. Master's thesis, TU München, 2014.

[KM02] Kevin Knight and Daniel Marcu. Summarization Beyond Sentence Extraction: A Probabilistic Approach to Sentence Compression. *Artif. Intell.*, 139(1):91–107, July 2002.

[KM03] Dan Klein and Christopher D. Manning. Accurate Unlexicalized Parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ACL '03, pages 423–430, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[KNS14] Prateek Karandikar, Matthias Niewerth, and Philippe Schnoebelen. On the State Complexity of Closures and Interiors of Regular Languages with Subwords. *CoRR*, abs/1406.0690, 2014.

[Koz96] Dexter Kozen. Kleene Algebra with Tests and Commutativity Conditions. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *LNCS*, pages 14–33. Springer Berlin Heidelberg, 1996.

[KPV02] Orna Kupferman, Nir Piterman, and MosheY. Vardi. Pushdown Specifications. In Matthias Baaz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 2514 of *Lecture Notes in Computer Science*, pages 262–277. Springer Berlin Heidelberg, 2002.

[KRS09] Jui-Yi Kao, Narad Rampersad, and Jeffrey Shallit. On NFAs Where All States are Final, Initial, or Both. *Theoretical Computer Science*, 410(47-49):5010 – 5021, 2009.

[KS14] Prateek Karandikar and Philippe Schnoebelen. On the State Complexity of Closures and Interiors of Regular Languages with Subwords. In *Descriptional Complexity of Formal Systems - 16th International Workshop, DCFS 2014, Turku, Finland, August 5-8, 2014. Proceedings*, pages 234–245, 2014.

[KT92] András Kornai and Zsolt Tuza. Narrowness, Pathwidth, and their Application in Natural Language Processing. *Discrete Applied Mathematics*, 36(1):87 – 92, 1992.

[KT10] Eryk Kopczynski and Anthony Widjaja To. Parikh images of grammars: Complexity and applications. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*, pages 80–89, 2010.

[Kui97] W. Kuich. *Handbook of Formal Languages*, volume 1, chapter 9: Semirings and Formal Power Series: Their Relevance to Formal Languages and Automata, pages 609 – 677. Springer, 1997.

[LB02] Edward Loper and Steven Bird. NLTK: The Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics-Volume 1*, pages 63–70. Association for Computational Linguistics, 2002.

[Ler05] Jérôme Leroux. A Polynomial Time Presburger Criterion and Synthesis for Number Decision Diagrams. In *Proceedings of LICS 2005*, pages 147–156, 2005.

[LL09] Martin Lange and Hans Leiß. To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm. *Informatica Didactica*, 8, 2009.

[LP09] Jérôme Leroux and Gérald Point. TaPAS: The Talence Presburger Arithmetic Suite. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 182–185, 2009.

[LRSS13] G.L. Litvinov, A.Ya. Rodionov, S.N. Sergeev, and A.N. Sobolevski. Universal Algorithms for Solving the Matrix Bellman Equations over Semirings. *Soft Computing*, 17(10):1767–1785, 2013.

[LS13] M. Luttenberger and M. Schlund. Convergence of Newton's Method over Commutative Semirings. In *LATA*, volume 7810 of *LNCS*, pages 407–418, 2013.

[LS14] Michael Luttenberger and Maximilian Schlund. Regular Expressions for Provenance. In *6th Workshop on the Theory and Practice of Provenance, TaPP'14, Cologne, Germany, June 12-13, 2014*, 2014.

[LS15] Michael Luttenberger and Maximilian Schlund. Convergence of Newton's Method over Commutative Semirings. *Information and Computation*, 2015.

[Lut10] Michael Luttenberger. *Solving Systems of Polynomial Equations: A Generalization of Newton's Method*. PhD thesis, Technische Universität München, 2010.

[Mil56] George A Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information. *Psychological Review*, 63(2):81, 1956.

[MN01] Mehryar Mohri and Mark-Jan Nederhof. Regular Approximation of Context-Free Grammars through Transformation. In *Robustness in Language and Speech Technology*, volume 17 of *Text, Speech and Language Technology*, pages 153–163. 2001.

[Moh02] M. Mohri. Semiring Frameworks and Algorithms for Shortest-Distance Problems. *J. Autom. Lang. Comb.*, 7(3):321–350, 2002.

[MS72] A. R. Meyer and L. J. Stockmeyer. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space. In *Proceedings of the 13th Annual Symposium on Switching and Automata Theory (Swat 1972)*, SWAT '72, pages 125–129, Washington, DC, USA, 1972. IEEE Computer Society.

[MS99] Christopher D. Manning and Hinrich Schütze. *Foundations of statistical natural language processing*, volume 999. MIT Press, 1999.

[MSSY01] Alexandru Mateescu, Arto Salomaa, Kai Salomaa, and Sheng Yu. A Sharpening of the Parikh Mapping. *RAIRO - Theoretical Informatics and Applications*, 35:551–564, 11 2001.

[MW67] J. Mezei and J.B. Wright. Algebraic Automata and Context-Free Sets. *Information and Control*, 11(1):3 – 29, 1967.

[Mü96] Haiko Müller. On Edge Perfectness and Classes of Bipartite Graphs. *Discrete Mathematics*, 149(1–3):159 – 187, 1996.

[Niv05] Joakim Nivre. Dependency grammar and dependency parsing. Technical report, 2005.

[Okh10] Alexander Okhotin. On the State Complexity of Scattered Substrings and Superstrings. *Fundam. Inform.*, 99(3):325–338, 2010.

[Orl77] James Orlin. Contentment in Graph Theory: Covering Graphs with Cliques. *Indagationes Mathematicae (Proceedings)*, 80(5):406 – 424, 1977.

[PBTK06] Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. Learning Accurate, Compact, and Interpretable Tree Annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 433–440. Association for Computational Linguistics, 2006.

[Pet99] I. Petre. Parikh's Theorem Does not Hold for Multiplicities. *J. Autom. Lang. Comb.*, 4(1):17–30, 1999.

[Pil73] D. L. Pilling. Commutative regular equations and Parikh's theorem. *Journal of the London Mathematical Society*, pages 663–666, 1973.

[PSS12] C. Pivoteau, B. Salvy, and M. Soria. Algorithms for Combinatorial Structures: Well-Founded Systems and Newton Iterations. *J. Comb. Theory, Ser. A*, 119(8):1711–1773, 2012.

[RM08] Anna N. Rafferty and Christopher D. Manning. Parsing Three German Treebanks: Lexicalized and Unlexicalized Baselines. In *Proceedings of the Workshop on Parsing German*, PaGe '08, pages 40–46, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[Ros11] Nathan E. Rosenblum. *The Provenance Hierarchy of Computer Programs*. PhD thesis, Madison, WI, USA, 2011. AAI3488666.

[RSJM05] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1–2):206–263, October 2005. Special Issue on the Static Analysis Symposium 2003.

[RSX12] Narad Rampersad, Jeffrey Shallit, and Zhi Xu. The Computational Complexity of Universality Problems for Prefixes, Suffixes, Factors, and Subwords of Regular Languages. *Fundam. Inform.*, 116(1-4):223–236, 2012.

[RVG07a] Ines Rehbein and Josef Van Genabith. Evaluating Evaluation Measures. In *NODALIDA*, pages 372–379, 2007.

[RvG07b] Ines Rehbein and Josef van Genabith. Treebank Annotation Schemes and Parser Evaluation for German. In *EMNLP-CoNLL*, pages 630–639, 2007.

[Sal90] A. Salomaa. Formal Languages and Power Series. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 103–132. Elsevier, Amsterdam, 1990.

[Sam00] Geoffrey Sampson. A Proposal for Improving the Measurement of Parse Accuracy. *International Journal of Corpus Linguistics*, 5(1):53–68, 2000.

[SB11] Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, volume 6702 of *Lecture Notes in Computer Science*, pages 245–251. Springer Berlin Heidelberg, 2011.

[SLE14] Maximilian Schlund, Michael Luttenberger, and Javier Esparza. Fast and Accurate Unlexicalized Parsing via Structural Annotations. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2014, April 26-30, 2014, Gothenburg, Sweden*, pages 164–168, 2014.

[SP81] M. Sharir and A. Pnueli. *Program Flow Analysis: Theory and Applications*, chapter 7: Two Approaches to Interprocedural Data Flow Analysis, pages 189–233. Prentice-Hall, 1981.

[SRJ] Stefan Schwoon, Thomas Reps, and Somesh Jha. Weighted Pushdown Systems Library. `http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/wpds/`.

[STK$^+$13] Djamé Seddah, Reut Tsarfaty, Sandra Kübler, Marie Candito, Jinho D. Choi, Richárd Farkas, Jennifer Foster, Iakes Goenaga, Koldo Gojenola, Yoav Goldberg, Spence Green, Nizar Habash, Marco Kuhlmann, Wolfgang Maier, Joakim Nivre, Adam Przepiorkowski, Ryan Roth, Wolfgang Seeker, Yannick Versley, Veronika Vincze, Marcin Woliński, Alina Wróblewska, and Eric Villemonte de la Clérgerie. Overview of the SPMRL 2013 Shared Task: A Cross-Framework Evaluation of Parsing Morphologically Rich Languages. In *Proceedings of the 4th Workshop on Statistical Parsing of Morphologically Rich Languages: Shared Task*, Seattle, WA, 2013.

[STL13] Maximilian Schlund, Michal Terepeta, and Michael Luttenberger. Putting Newton into Practice: A Solver for Polynomial Equations over Semirings. In *LPAR*, pages 727–734, 2013.

[Str52] A. N. Strahler. Hypsometric (area-altitude) analysis of erosional topology. *Geological Society of America Bulletin*, 63(11):1117–1142, 1952.

[STW08] Anish Das Sarma, Martin Theobald, and Jennifer Widom. Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 1023–1032, 2008.

[Sud04] M. Suderman. Pathwidth And Layered Drawings Of Trees. *Int. J. Comput. Geometry Appl.*, 14(3):203–225, 2004.

[Tar81] Robert Endre Tarjan. A Unified Approach to Path Problems. *J. ACM*, 28(3):577–593, 1981.

[Tes53] L. Tesnière. *Esquisse d'une syntaxe structurale*. C. Klincksieck, 1953.

[THK$^+$03] Heike Telljohann, Erhard W. Hinrichs, Sandra Kübler, Heike Zinsmeister, and Kathrin Beck. Stylebook for the Tübingen Treebank of Written German (TüBa-D/Z). Seminar für Sprachwissenschaft, Universität Tübingen, Germany, 2003.

[UNMT06] Yuya Unno, Takashi Ninomiya, Yusuke Miyao, and Jun'ichi Tsujii. Trimming CFG Parse Trees for Sentence Compression Using Machine Learning Approaches. In *Proceedings of the COLING/ACL on Main Conference Poster Sessions*, COLING-ACL '06, pages 850–857, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics.

[vL78] Jan van Leeuwen. Effective Constructions in Well-Partially-Ordered Free Monoids. *Discrete Mathematics*, 21(3):237 – 252, 1978.

[VSS05] K. N. Verma, H. Seidl, and T. Schwentick. On the Complexity of Equational Horn Clauses. In *CADE*, volume 3632 of *LNCS*, pages 337–352, 2005.

[VT13] Naveneetha Vasudevan and Laurence Tratt. Detecting Ambiguity in Programming Language Grammars. In *Software Language Engineering*, volume 8225 of *LNCS*, pages 157–176. 2013.

[VV] Vesal Vojdani and Varmo Vene. Goblint. `http://goblint.in.tum.de/`.

[WACL05] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 97–118, Berlin, Heidelberg, 2005. Springer-Verlag.

[WB95] Pierre Wolper and Bernard Boigelot. An Automata-Theoretic Approach to Presburger Arithmetic Constraints (Extended Abstract). In *Proceedings of SAS '95*, pages 21–32, London, UK, UK, 1995. Springer-Verlag.

[WE07] Dominik Wojtczak and Kousha Etessami. PReMo : An Analyzer for Probabilistic Recursive Models. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 66–71, 2007.

[YK01] Kenji Yamada and Kevin Knight. A syntax-based statistical translation model. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, ACL '01, pages 523–530, Stroudsburg, PA, USA, 2001. Association for Computational Linguistics.

[Zet15] Georg Zetzsche. An Approach to Computing Downward Closures. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, pages 440–451, 2015.

[Zwi02] Uri Zwick. All Pairs Shortest Paths Using Bridging Sets and Rectangular Matrix Multiplication. *J. ACM*, 49(3):289–317, 2002.