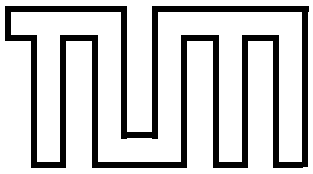


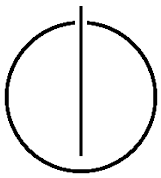
FAKULTÄT FÜR INFORMATIK

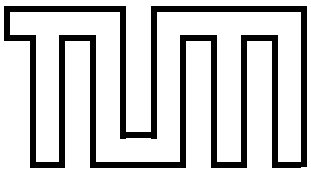
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Symbolic Representations of Semilinear Sets

Michael Kerscher





FAKULTÄT FÜR INFORMATIK

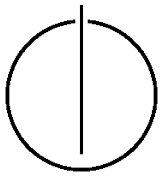
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Symbolic Representations of Semilinear Sets

Symbolische Darstellungen von semilinearen Mengen

Author:	Michael Kerscher
Supervisor:	Univ.-Prof. Dr. Dr. h.c. Javier Esparza
Advisor:	Dipl.-Math. Maximilian Schlund
Date:	June 16, 2014



I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, June 16, 2014 Michael Kerscher

Acknowledgments

I would like to thank my supervisor Prof. Javier Esparza for giving me the chance to write my thesis at his chair and accepting to supervise my thesis. Additionally I would like to express my gratitude to my advisor Maximilian Schlund who introduced me to this interesting topic, gave me lots of helpful comments and advice, always had time for my concerns, and for his endless patience. Furthermore, I would also thank Michael Luttenberger who also gave me helpful comments.

Moreover, I am very grateful to Leonhard Rabel who proof-read this thesis, always listened to my problems and for being a really good friend for many years now. Also I would like to thank my fellow students, especially Marek Kubica, Lars Hupel and Paul Emmerich for lots of discussion and them helping me with programming problems.

Of course I also want to thank the many friends who accompanied me over many years, where I especially want to mention Bertram Wermuth and Stefan Huber who always had an open ear for me when I needed them.

At last I want to thank my family for their unconditional support over all the years and their understanding if I sometimes had only little time for them.

Abstract

In this thesis, we analyse how number decision diagrams can be endowed with a semiring structure. To show this we define addition and multiplication on these automata and as well as the Kleene star on semilinear sets so they can be used for the computation of fixed point equations with the newton method in the FPsolve tool. As the explicit representation of semilinear sets has several disadvantages, we translate them into special automata called number decision diagrams which can be used to efficiently encode Presburger definable sets. We define addition and multiplication on these automata by only using high-level automata operations like intersection, union and projection. This number decision diagram representation has several advantages over the representation as semilinear sets, as these automata have a canonical representation which allows for efficient equality and membership tests. At last we also show a method which we suppose to extract a superset of the constants of the semilinear sets from such an automaton.

Zusammenfassung

In dieser Arbeit zeigen wir, wie *number decision diagrams* (NDDs) mit einer Semiringstruktur ausgestattet werden können. Dazu definieren wir Addition und Multiplikation auf diesen Automaten. Zusätzlich definieren wir den Kleene Stern Operator auf Semilinearen Mengen, um diese bei der Berechnung von Fixpunktgleichungen mit Newton- und Kleenelösern in dem Tool FPsolve verwenden zu können. Da die explizite Representation der Semilinearen Mengen mehrere Nachteile besitzt, transformieren wir diese in die NDD Darstellung, welche spezielle endliche deterministische Automaten sind und dazu genutzt werden können, Presburger definierbare Mengen effizient zu kodieren. Auf diesen Automaten definieren wir dann nur unter Verwendung von höheren Automatenoperationen wie Schnitt, Vereinigung und Projektion die Addition und Multiplikation. Die NDD Darstellung bietet einige Vorteile, da diese Automaten eine kanonische minimale Darstellung besitzen, was es ermöglicht, effiziente Gleichheits- und Teilmengentests durchzuführen. Zum Schluss zeigen wir ausserdem einen Algorithmus, der vermutlich eine Obermenge von Konstanten der Semilinearen Menge aus einem solchen Automaten extrahieren kann.

Contents

Outline of the Thesis

Part I: Introduction and Theory

CHAPTER 1: INTRODUCTION

This chapter introduces the topic of the thesis and the current state.

CHAPTER 2: THEORETICAL FOUNDATION

The theoretical foundation of the thesis is presented here.

CHAPTER 3: MATHEMATICAL THEORY

Mathematical theory used in the algorithms is explained here.

Part II: Algorithms and real Implementation

CHAPTER 4: ALGORITHMS

In this chapter the algorithms needed to implement the theoretical work are described.

CHAPTER 5: IMPLEMENTATION AND OPTIMIZATIONS

The frameworks used and implementation are mentioned here. In addition the chapter describes optimizations used in the implementation.

CHAPTER 6: CONCLUSION

We conclude the thesis.

CHAPTER 7: FURTHER WORK

Here we will talk about further work.

Part I.

Introduction and Theory

1. Introduction

1.1. Motivation

In this thesis we study symbolic representations of semilinear sets and their possible usage in tools like FPsolve[STL13]. FPsolve uses semirings to approximate the solutions of fixed point equations with Newton's and Kleene's method. They have to support efficient operations to get specific information from the solution. For this we want to use semilinear sets to represent the input. To be able to work with these semilinear sets, we have to endow them with a semiring structure to be able to solve the equations on them. But as some operations on semilinear sets like the multiplication are not efficient and do not support efficient queries like testing if a vector is included in the set, we want to transform these sets to number decision diagrams (NDDs). These NDDs are a canonical representation of sets of numbers and they enable us to easily check for equality of two sets or the inclusion of one set in the other. We also want to be able to check if a given vector is included in the set. Therefore we show that these NDDs can be endowed with a semiring structure and used in tools like FPsolve.

One of our design goals during the implementation was to be independent of the underlying automata data structures. For this reason we use the GENEPI framework, which serves as an abstraction layer between our implementation and the actual automata library. All our algorithms are designed to be independent of the used data structures to be able to easily switch from an automata representation to other symbolic representations like Presburger formulae. Some of the algorithms might seem to be overly complicated due to the fact that we do not have access to the internal data structure but most of them have similar complexity than what we would get if we had access.

1.2. Previous and Related Work

There have already been implementations of semilinear sets in FPsolve, but in contrast to this work the multiplication and Kleene star operation required ex-

ponential space which led to the implementation of an over-approximation with multilinear sets [STL13]. Additionally, some kind of queries are not possible in an efficient way. For example there is no efficient method to check whether a vector is included in a set without solving a NP-complete problem. Moreover, the representation grows exponentially during multiplication and application of the Kleene star, and the minimizing process to filter redundant vectors also requires solving NP-complete problems[STL13].

There is an ongoing research on extracting semilinear sets from NDDs, but at the moment the known algorithms are only operating on specific subsets of semilinear sets, or are complex and cause blow up in the semilinear set representation. In [Ler05] Leroux describes a method for deciding in polynomial time whether a least significant digit first NDD represents a Presburger-definable set. Moreover, he provides an algorithm which computes, in polynomial time, a Presburger-formula that defines the set represented by the NDD. For a special case of semilinear sets, Lugiez characterizes these semilinear sets, where each linear set has the same periods and computes such sets from an automaton representing them in double exponential time in [Lug04][Lug05]. Additionally, an algorithm for computing a quantifier-free formula from an automaton representing the integer elements of a polyhedron is described by Latour in [Lat05].

But, as these results provide too complicated, and more important, too specific algorithms for extracting semilinear sets from an automaton representation and these extracted sets are often blown up, we will stay in the NDD representation and do not want to convert between these representations in our algorithms.

2. Foundations

2.1. FPsolve

To solve a fixed point equation $x = f(x)$ over ω-continuous semirings, one can compute the sequence $0, f(0), f^2(0), \ldots$ which converges to the least fixed point μf[Kle52] which is the supremum of the sequence $0 \leq f(0) \leq f^2(0) \leq \ldots$. This approach is known as Kleene's method and it's convergence can be accelerated if the underlying semiring is commutative. [EKL07] show that Newton's method and the Hopkin-Kozen acceleration are instances of a general algorithm for computing the least fixed point over arbitrary ω-continuous semirings. For idempotent semirings they also show that their approach reaches μf after n iterations where n is the number of equations[EKL07].

This result was used in [EKL10] to provide a new generic approach to solve dataflow equations used in interprocedural dataflow analysis. They use ω-continuous semirings instead of complete semilattices, which are commonly used in program analysis frameworks. Also this new approach has the advantage that, contrary to Kleene's method, Newton's method always terminates for arbitrary idempotent and commutative semirings and gets an upper bound of n iterations for n equations. Further analysis in [LS13] gives a lower bound for the convergence of Newton's method over commutative semirings.

FPsolve[FPs14] implements these results and is described in [STL13]. This C++-framework is an implementation of Newton's method on arbitrary ω-continuous semirings. FPsolve is easily extended with new semirings which can be used in the solver. To this end every semiring has to provide at least an addition, a multiplication and a Kleene star method. There are already some commonly used semirings available like the float semiring, the free semiring, the prefix semiring, the tropical semiring and a semiring for commutative regular expressions. This thesis also provides an implementation of the semilinear sets represented by NDDs which can be used in the solver.

2.1.1. Semirings

The following definition of semirings and further information can also be found in [DKV09].

A *monoid* is a non-empty set M with an associative binary operation $\odot$ on M and a neutral element 1 with $m \odot 1 = 1 \odot m = m$ for all $m \in M$. We write $\langle M, \odot, 1 \rangle$ for such a monoid. A monoid is *commutative* if $m_1 \odot m_2 = m_2 \odot m_1$ for all $m_1, m_2 \in M$. A monoid is *idempotent* if $m \odot m = m$ for all $m \in M$.

A *semiring* is a set S with two binary operations $\oplus$ and $\otimes$ and two elements 0 and 1 such that:

1. $\langle S, \oplus, 0 \rangle$ is a commutative monoid
2. $\langle S, \otimes, 1 \rangle$ is a monoid
3. distributivity laws $(a \oplus b) \otimes c = a \otimes c \oplus b \otimes c$ and $c \otimes (a \oplus b) = c \otimes a \oplus c \otimes b$ hold for all $a, b, c \in S$
4. $0 \otimes a = a \otimes 0 = 0$ for all $a \in S$

We write $\langle S, \oplus, \otimes, 0, 1 \rangle$ for such a semiring. A semiring is *idempotent* if $\langle S, \oplus, 0 \rangle$ is idempotent. It is *commutative* if $a \otimes b = b \otimes a$ for all $a, b \in S$.

2.1.1.1. ω-continuous Semirings

For further information, these definitions of ω-continuous semirings are also presented in [Kui97] and [EKL07].

A semiring S is

- *complete* if the regular finite sums are extended to allow the definition of infinite sums which are associative, commutative and distributive.
- *naturally ordered* if the binary relation $\leq$ given by $a \leq b \Leftrightarrow \exists c. a \oplus c = b$ is a partial order.
- *ω-continuous* iff S is complete, naturally ordered and $\sup \{\sum_{i=0}^{n} a_i | n \in \mathbb{N}\} = \sum_{i \in \mathbb{N}} a_i$ is satisfied.

Example Let $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$ then $\langle \mathbb{N}^\infty, \oplus, \otimes, 0, 1 \rangle$ is an ω-continuous semiring.

A semiring can be viewed as a ring without subtraction. An example for a semiring is the set of natural numbers $\mathbb{N}$. There are also other important semirings, which are used in the newton solver like the *tropical semiring* $\langle \mathbb{N}^\infty, \min, \oplus, \infty, 0 \rangle$. And in this work we describe how a semiring structure can be found on semilinear sets.

2.2. Semilinear Sets

The definition of linear and semilinear sets can be found together with some proofs in [Gin66, pp.143-144] or [Lat05].

Definition $L \subseteq \mathbb{N}^n$ is called a *linear set* if there is a *constant* $c_0 \in \mathbb{N}^n$ and a finite number of *periods* $p_i \in \mathbb{N}^n$ such that $L = c_0 + \sum_{i=1}^{m} \mathbb{N} p_i$. We will also write this as $L(c_0; p_1, \ldots, p_m)$. The set of periods $\{p_1, \ldots, p_m\}$ can also be written as P. This allows us to use this short notation $L(c; p_1, \ldots, p_m) = L(c; P)$.

Example The linear set $L((1,2); (0,2), (3,0))$ defines the set $L = \{(0,0) + k_1(0,2) + k_2(3,0) | k_1, k_2 \in \mathbb{N}\} = \{(1 + 3k_2, 2 + 2k_1) | k_1, k_2 \in \mathbb{N}\}$ (see figure 2.1).

Definition $S \subseteq \mathbb{N}^n$ is called a *semilinear set* if it is a finite union of linear sets: $S = L_1(c_1; p_{1_1}, \ldots, p_{1_n}) \cup \ldots \cup L_m(c_m; p_{m_1}, \ldots, p_{m_n})$. A finite union of semilinear subsets of $\mathbb{N}^n$ is again semilinear.

2.3. Parikh's Theorem

Let $\Sigma = \{a_i \ : 1 \leq i \leq n\}$ and let $L \subseteq \Sigma^*$ be a context-free language. Then we can define a function $\Phi : L \to \mathbb{N}^n$ which counts the occurrences of each a_i and returns this number on the i-th component of the resulting vector. Let $n = 4$ and $w = a_1 a_2 a_1 a_4$ then $\Phi(w) = (2, 1, 0, 1)$. $\Phi(L)$ is called a commutative image of L and is a semilinear subset of $\mathbb{N}^n$[Par66]. More information on Parikh images can also be found in [Gin66].

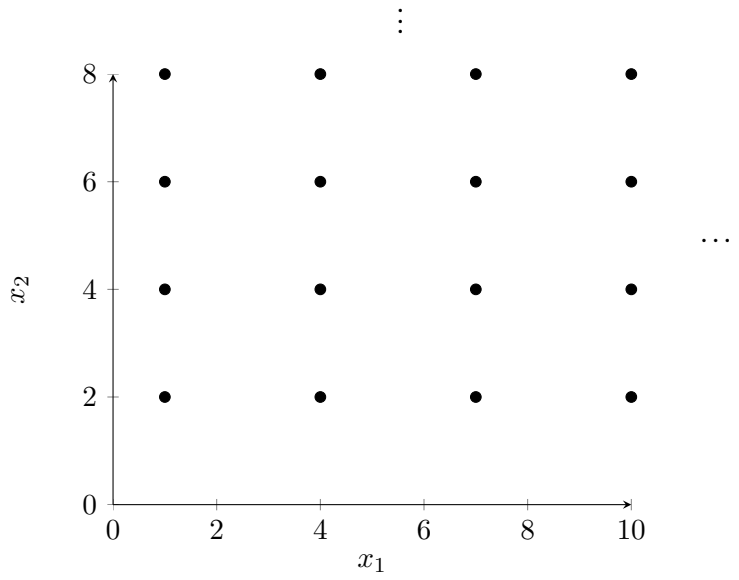


Figure 2.1.: The linear set $L = \{(1,2) + k_1(0,2) + k_2(3,0)\}$

Parikh's Theorem states that the Parikh image of any context-free language is semilinear and every semilinear set coincides with the Parikh image of some regular languages. It follows that every context-free language has the same Parikh image as some regular language. For example the language $\{a^n b^n | n \geq 0\}$ has the same Parikh image as $(ab)^*$. [EGKL11] describe the construction of a finite automaton which recognizes a regular language corresponding to a given context-free language. Parikh's proof is a constructive one, but the construction of the automaton is complicated and creates automata of size $\mathcal{O}(n^n)$. [EGKL11] achieves a much simpler and direct construction in $\mathcal{O}(4^n)$ for grammars in Chomsky normal form and a lower bound of $\Omega(2^n)$.

2.4. Number Decision Diagrams

In this work we use automata accepting sets of vectors of numbers in a binary encoding called *number decision diagrams*. We introduce the idea with vectors in $\mathbb{N}^1$ and, at the end of this section, we generalise this to vectors in $\mathbb{N}^k$. If we are going to accept a number $n \in \mathbb{N}^1$, we have to encode n in such a way that a deterministic finite automaton (DFA) can recognize it. Every natural number can be represented

in binary representation like $\langle n \rangle_2 = b_k \dots b_4 b_3 b_2 b_1 b_0$ with $n = \sum_{i=0}^{k} 2^i b_i$. A DFA can read this binary representation as an input word and therefore recognize the number n. This also works in other bases than 2 and only increases the automaton alphabet from $\Sigma = \{0, 1\}$ to $\Sigma = \{0, \dots, \text{base} - 1\}$. In addition it increases the number of transitions at each state.

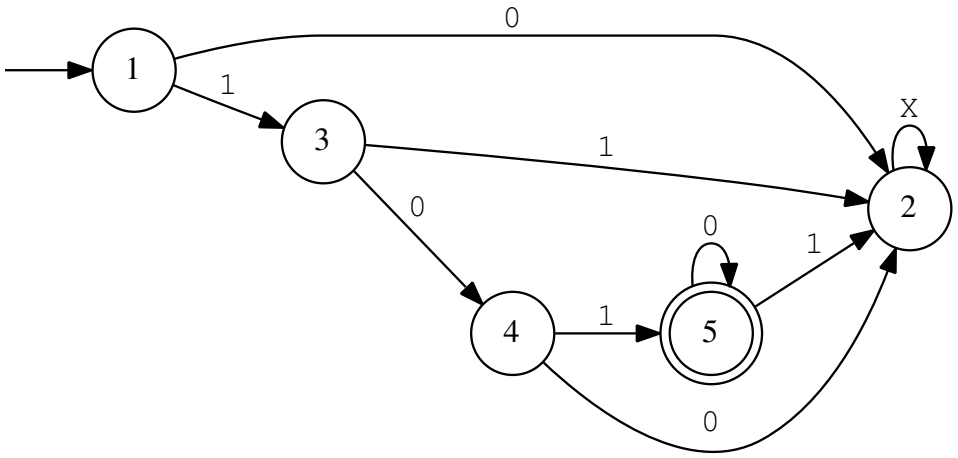


Figure 2.2.: Automaton accepting the number 5_{10} as the binary word 101_2 in LSBF encoding with the invalid state 2

There are two ways to read the binary representation with the automaton. In *MSBF* encoding we use the *most significant bit first*. There is no unique representation for a number in binary representation (e.g. $000101 \equiv 101 \equiv 0101$) and, therefore, we have to accept an arbitrary number of zeros at the start of the word. Because of this the automaton which accepts the binary encoding of $b_4 b_3 b_2 b_1 b_0$ has to accept all words $0^* b_4 b_3 b_2 b_1 b_0$. In *least significant bit first* (*LSBF*) encoding the automaton must recognize the words in $b_0 b_1 b_2 b_3 b_4 0^*$. The automaton in figure 2.2 reads the number 5 in LSBF as 101 and the language of the automaton is 1010^*.

2.4.1. Signed Numbers

There exist frameworks, which are working only on numbers in $\mathbb{N}$ (e.g. MONA), but there are also frameworks which can recognize all integers in $\mathbb{Z}$ like LASH. As there is no sign symbol $(+, -)$ available in the input alphabet, there also exists an encoding for negative numbers called *two's complement*. A N-bit number in *two's complement* is defined as $n = -2^{N-1} b_{N-1} + \sum_{i=0}^{N-2} 2^i b_i$. If the N-th bit is 0, then we only add up positive numbers and get a positive value. But if the N-th

bit is 1 we also add up -2^{N-1} which is always smaller than the rest of the term because $\forall b_i : 2^{N-1} > \sum_{i=0}^{N-2} 2^i b_i$. Intuitively, we can see this bit as a *sign bit*. In frameworks which accept signed integers (like LASH, see section 5.1.1) the first transition always encodes the sign bit in *MSBF*. In *LSBF* encoding the frameworks have to distinguish negative and positive numbers with the transition going to the final state. For positive numbers this 0 transition can be included in the self loop at the final state. For natural numbers we only have to accept 0^* from the accepting state, whereas for integer numbers we can only reach the accepting end state for negative numbers by a final transition with 1.

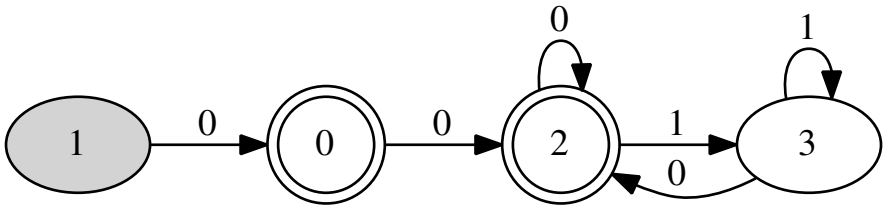


Figure 2.3.: Automaton accepting multiples of four in LSBF with sign bit

In figure 2.3 there is an automaton which accepts all positive numbers which are multiples of four $\{n|n = 4k, k \in \mathbb{N}\}$ in LSBF encoding. A multiple of four in binary representation always has the bit sequence 001 as a prefix and then there is an arbitrary amount of 0's and 1's. In LSBF encoding with a sign bit we always have at least one 0 standing for the $+$-sign at the end of the word. This is the transition from state 3 to state 2. It cannot accept a word ending with 1 because that would encode a negative number.

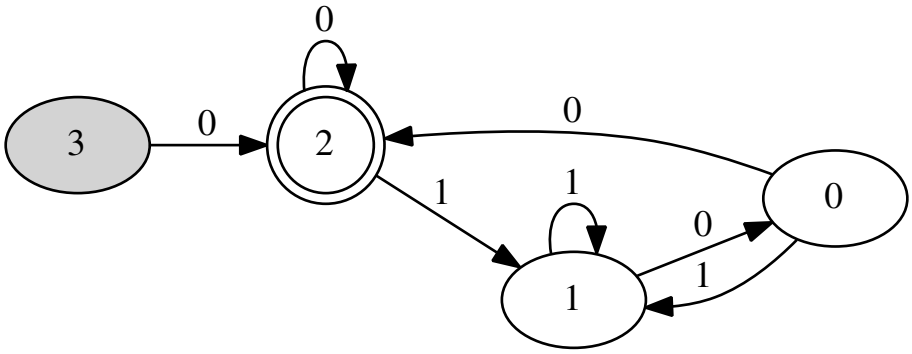


Figure 2.4.: Automaton accepting multiples of four in MSBF with sign bit

Figure 2.4 shows an automaton accepting the same set $\{n|n = 4k, k \in \mathbb{N}\}$ but in MSBF encoding. It first reads the sign bit 0 and then read an arbitrary amount of

0's and 1's which is mostly done in the loop between the states 0 and 1 and we reach the accepting state only if we have found the postfix 100 in the word which marks multiples of four.

2.4.2. Vectors of Numbers

In the previous section we explained how NDDs can recognize vectors $\vec{v} \in \mathbb{N}^1$, but they can also accept vectors $\vec{v} \in \mathbb{N}^k$. The idea is to read one bit simultaneously of all k components, which can be seen for $k = 2$ in figure 2.5. There the example automaton has to read the input vectors $\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ representing the vector $\begin{bmatrix} 5 \\ 6 \end{bmatrix}$ in LSBF to reach the accepting state 5. We will use an X in a vector notation like $\begin{bmatrix} 1 \\ X \end{bmatrix}$ throughout this thesis to denote that we can accept every $v \in \Sigma$ for the X component and could also write $\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}$.

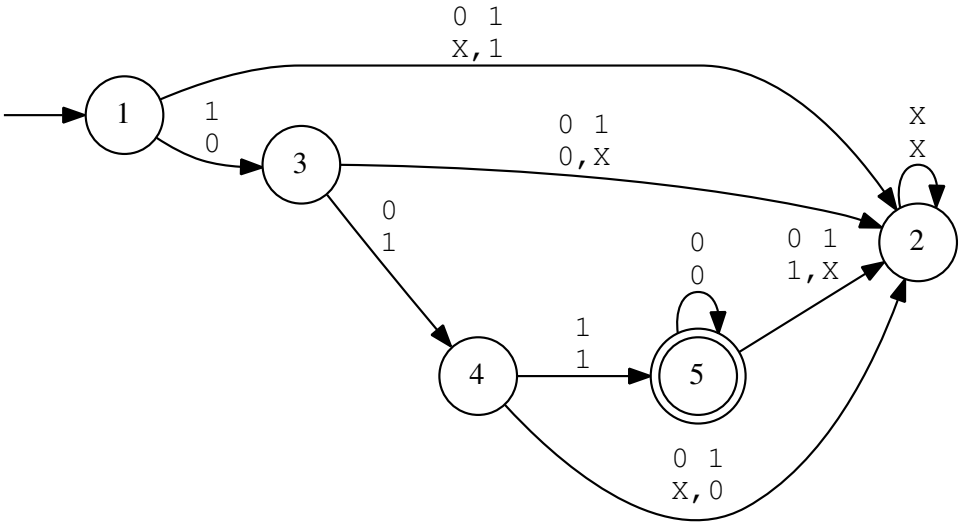


Figure 2.5.: Automaton accepting the vector $(5, 6)^T \in \mathbb{N}^2$

2.4.3. Minimality and Canonicity of Number Decision Diagrams

Another interesting property of NDDs is their minimality, which is the reason that some operations are very efficient like checking two NDDs for equality. As NDDs are deterministic finite automata, there is an algorithm which minimizes a given automaton A to a minimal one, which is linear in $\mathcal{O}(|E| \log |Q|)$ with E being the transitions of A and Q the states of A. A DFA is minimal if there is no other DFA which is smaller in the number of states recognizing the same language[DKV09]. After each operation on these automata, which could create non-minimal automata, these NDDs can always be minimized. There is only one minimal automaton for a given language, therefore we can use this form as the canonical representation for a set of vectors. If two automata accept the same language and therefore the same set of vectors we know that they have the same minimal canonical automaton, which can be checked in linear time regarding the size of the automaton. Another advantage of this canonicity is that this representation is always the most memory efficient representation possible.

2.5. Presburger Arithmetic

Presburger arithmetic is the first-order theory of natural numbers with addition and equality proposed by Presburger[Sta84]. Unlike Peano arithmetic, which has both the addition and multiplication, Presburger arithmetic lacks multiplication.

It can be shown that Presburger arithmetic is *consistent*, it is not possible to derive a statement and its negation from the axioms. Additionally, it is also *complete*, meaning it is possible to derive each statement in the language of Presburger arithmetic from the axioms or derive the negated statement. A third and very interesting property is its *decidability*, which means that for each statement it is possible to decide if it is a theorem. Peano arithmetic lacks these properties.

Presburger arithmetic formulae consist of several individual symbols like $0, 1, +$, $\exists, =, \neg$ and $\Rightarrow$, and also symbols for variables like $p, q, r, \ldots$ for predicates and $x, a, b, c, \ldots$ for integer variables. A variable which is not bound by a quantifier is called a *free variable*. Given a formula $\phi(x_1, x_2, \ldots, x_n)$, $x_1, x_2, \ldots, x_n$ are free variables in the Presburger formula, which we can assign to the formula. Bound variables are variables which are bound to a quantifier like x in $\phi(a) \equiv \exists x.x + x = a$.

A set A is a Presburger set in $\mathbb{N}^n$ if for some $P(x_1, x_2, \ldots, x_n)$ in the set of Presburger formulae it is definable as $A = \{(x_1, x_2, \ldots, x_n) | P(x_1, x_2, \ldots, x_n)\}$.

2.5.1. Connection to Semilinear Sets

Ginsburg proves in [GS66] that the family of semilinear sets is identical with the family of sets defined by Presburger formulae and one representation can be calculated from the other. All results connected to Presburger arithmetic can therefore also be used in connection with semilinear sets.

2.5.2. Connection to Automata / Number Decision Diagrams

Boudet and Comon devised an algorithm to efficiently compute an automaton which accepts the set of solutions of a linear Diophantine equation[BC96]. Their method can be shown to be almost optimal, as for the existential fragment of Presburger arithmetic, which is NP-complete, it runs in exponential time. In addition, it runs in triply exponential time in the size of the formula for the whole Presburger arithmetic, which is known to be complete for double exponential space[FR79]. They describe a method for recognizing the solutions of linear Diophantine equations, inequations and also systems of equations. The latter have the same complexity as the former in the worst case, but are better suited in the practice.

Other related work was done by Muchnik, who devised a way do determine given an automaton recognizing a set of numbers whether this set is Presburger definable[Muc03].

2.5.3. Automata to Semilinear Sets

While it is possible to convert automata to semilinear sets, there is no easy way how to determine the corresponding semilinear set given an automaton which works on the binary representation of sets of numbers. This transformation from automata to semilinear sets is interesting because the automata have a canonical form, while Presburger formulae and semilinear sets lack this canonicity. Also some operations on automata are efficient whereas doing them on formulae or semilinear sets directly is inefficient. For example querying if a given vector is included in a semilinear set $v \in S$, while the same query on an automaton would be linear in the size of v. But some operations are much easier when the set is represented in one of these explicit forms like in the semilinear set representation or as a formula. For example doing a translation by some vector a on the formula $\{c + k_1p_1 + k_2p_2 | c, p_1, p_2 \in \mathbb{N}^n, k \in \mathbb{N}\}$ is easier on formulae than on automata. The reason for this is that we can just add it to the constant c in the formula and get

$\{c + a + k_1p_1 + k_2p_2 | a, c, p_1, p_2 \in \mathbb{N}^n, k \in \mathbb{N}\}$, while we would have to perform a multiplication on two automata.

Muchnik characterizes the automata that represent Presburger formulae, but there is no simple way to extract the semilinear set from this characterization. Lugiez describes an algorithm in [Lug04][Lug05] which can extract a semilinear representation in double exponential time for semilinear sets of the form $L(C, P) = \bigcup_{c \in C} L(c, P)$ where all linear sets share the same set of periods.

Some other related work was done by Latour, who describes an algorithm which could extract quantifier-free formulae that represent the set of integer vectors accepted by the automaton[Lat04]. His implementation even worked with automata with more than 10000 states, but they only use automata accepting the encodings (MSBF encoding) of the natural solutions of systems of linear Diophantine inequations, i.e. convex polyhedra in $\mathbb{N}^n$. Latour could also extract a quantifier-free formula from a given automaton if this automaton recognized the integer elements of a polyhedron[Lat05]. In [Ler05] Leroux found a method for deciding if an NDD represents a Presburger-definable set and computes from this characterization the corresponding Presburger formula that defines this set[Ler05].

3. Mathematical Theory

3.1. Representing Semilinear Sets with Number Decision Diagrams

We already mentioned in section 2.4.3 that NDDs have a canonical form which enables us to do some operations much easier. The NDDs can efficiently be constructed from Presburger formulae or semilinear sets. As these automatons can only represent the binary encodings of the original set and we only use operations under which the automaton representation is closed we again get only automatons which represent binary encodings of some semilinear set.

3.2. Endowing Semilinear Sets and Number Decision Diagrams with a Semiring Structure

To use the semilinear sets respectively NDDs to calculate fix points we have to endow them with a semiring structure. We want to get an idempotent commutative ω-continuous semiring, so we must define an addition, multiplication and star operation. First, we will define these operations on the semilinear sets and then on the automata representation.

3.2.1. Representing Semiring Elements

We already have a semilinear set which we got as input and has to be transformed to an NDD. Assume we have a semilinear set S which is a finite union of linear sets $S = \bigcup_i L(c_i p_1, \ldots, p_{m_i})$ with $c_i, p_{i_j} \in \mathbb{N}^n, i \in [m], j \in [m_i]$. In this section we first explain how we can transform a semilinear set to a Presburger formula and then transform the formula to a system of linear equations. These linear equation are then used to construct an NDD.

Our example in this case is the regular expression $abbc(ab)^*(cc)^*$, which has the Parikh image $L((1,2,1);(1,1,0),(0,0,2))$. We can rewrite this into a Presburger

formula: $\Psi(\vec{x}, \mu, \lambda) \equiv \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} + \mu \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} + \lambda \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix} = \vec{x}$. The goal is to construct an automaton which will recognize all solutions $\vec{x}$ to satisfy $\Psi(\vec{x}, \mu, \lambda)$.

It is possible to create NDDs that accept the solutions of linear equations as their language by creating an automaton for each equation and then intersect all of them. Given a constant $c = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \in \mathbb{N}^n$ we can construct a system of linear equations in a straight forward way.

$$\begin{aligned} x_1 + \ 0 + \ 0 &= 1 \\ 0 + x_2 + \ 0 &= 2 \\ 0 + \ 0 + x_3 &= 1 \end{aligned}$$

To include the periods of our set, we also have to include the μ and λ into the system and get for our example $L((1,2,1);(1,1,0),(0,0,2))$:

$$\begin{aligned} x_1 + \ 0 + \ 0 - 0 - \ 0 &= 1 \\ 0 + x_2 + \ 0 - \mu - \ 0 &= 2 \\ 0 + \ 0 + x_3 - \mu - 2\lambda &= 1 \end{aligned}$$

This system defines all solutions of $\Psi\vec{x}, \mu, \lambda$, but we want all vectors $\vec{x}$ for all possible $\mu \in \mathbb{N}$ and $\lambda \in \mathbb{N}$. So we introduce an existential quantifier for these variables and get $\Phi(\vec{x}) \equiv \exists \mu, \lambda \in \mathbb{N}. \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} + \mu \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} + \lambda \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix} = \vec{x}$.

To create the automaton which recognizes the set $\{\vec{x} | \Phi(\vec{x})\}$ we first create an automaton for each of the equations of our system of linear equations with an algorithm which is described in section 3.2.1.1. Next, we compute the intersection of these automata to recognize only the solutions which fulfill all the equations. If we have n linear equations, then we create an automaton A_i for the i-th equation and get our result automaton with $A = \bigcap_{i=1}^{n} A_i$. Now we have an automaton accepting

$L((x_1, x_2, x_3, \mu, \lambda)^T)$, but we are only interested in $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ so we project away both μ and λ which is analog to the introduction of the existential quantifier and get an automaton accepting $L(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix})$.

3.2.1.1. Constructing an Automaton from a Linear Equation

The construction of an automaton which recognizes a given linear equation is described in detail in [BC96]. The idea of the algorithm is to start at the original equation, which we use as the start state of our resulting automaton and create transitions to new states, which encodes a balance indicating how far we are from the end state. The following example appears in [BC96], where the linear equation is $x + 2y - 3z = 2$. We introduce a vector $\vec{b} = \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} \in \mathbb{N}^3$, which stands for the bits of the number which we have to read to get to the next state. The next state is calculated as $a_1x_1 + \ldots + a_nx_n = k \xrightarrow{\vec{b}} a_1x_1 + \ldots + a_nx_n = \frac{k-(b_1a_1+\ldots+b_na_n)}{2}$. For example if $\vec{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$, then we would get a transition from $x + 2y - 3z = 2 \xrightarrow{\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}} x + 2y - 3z = 1$. If $k - (a_1b_1 + \ldots + a_nb_b)$ is not even, then we add a transition from the current state to the fail state with $\vec{b}$. From this newly created, valid states we can again use the algorithm and generate new states. We have finished when there is no new state for which we have to calculate its transitions. The state with the constant 0 is then marked as the final state. The resulting automaton for this example can be seen in figure 3.1, where the number of the state indicates the right side of the equation.

An upper bound for the number of states and transitions for automatons generated by this algorithm is also stated in [BC96]. For an equation $\sum_{i=1}^{n} a_ix_i = k$, the number of states in the resulting automaton is at most $|k| + \sum_{i=1}^{n} |a_i| + 1$ and each state has at most 2^n transitions.

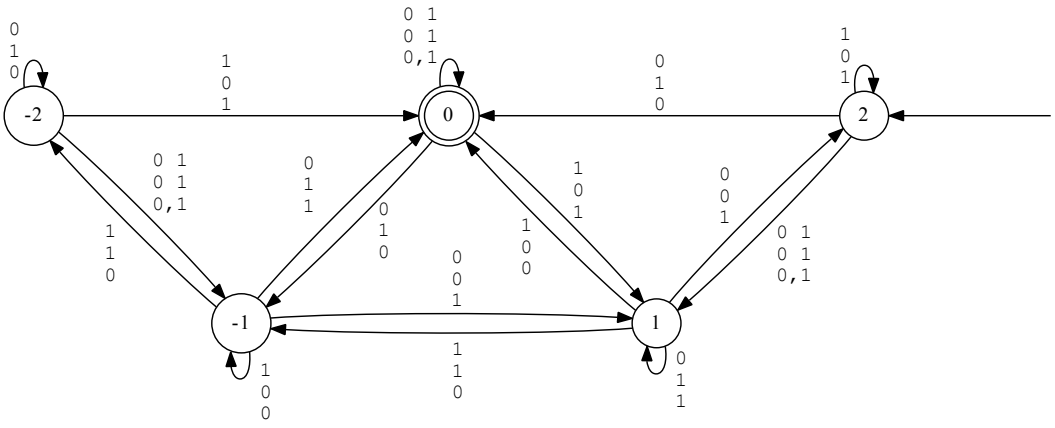


Figure 3.1.: The automaton for $x + 2y - 3z = 2$

3.2.2. Addition

Addition on semilinear sets is defined as the union of two semilinear sets. Given two semilinear sets S_1 and S_2, we get $S = S_1 \oplus S_2 = S_1 \cup S_2$[Gin66]. If we do a union on two automata, we get a resulting automaton, which recognizes words of both input automata.

Example For $S_1 = \{c_1; p_1, p_2\}, S_2 = \{c_2; p_3\}$ we get $S_1 \oplus S_2 = \{c_1; p_1, p_2\} \cup \{c_2; p_3\}$ (c_i are constants, p_i are periods).

3.2.3. Multiplication

Given two semilinear sets S_1 and S_2 we define multiplication as follows: $S = S_1 \otimes S_2 = \{L_1 \otimes L_2 | L_1 \in S_1, L_2 \in S_2\}$ ($L_{1,2}$ are linear sets) with $L_1 \otimes L_2 = \{v_1 + v_2 | v_1 \in L_1, v_2 \in L_2, v_{1,2} \in \mathbb{N}^k\}$ (with k the dimension of v and $+$ the vector addition in $\mathbb{N}^k$). This definition says that we add all vectors of S_1 with all vectors of S_2.

$$
\begin{aligned}
L_1(c_1; P_1) \otimes L_2(c_2, P_2) &= \left\{ c_1 + \sum_{i=0}^{j} m_i p_{1_i} | m_i \in \mathbb{N} \right\} \otimes \left\{ c_2 + \sum_{i=0}^{l} n_i p_{2_i} | n_i \in \mathbb{N} \right\} \\
&= \left\{ c_1 + c_2 + \sum_{i=0}^{j} m_i p_{1_i} + \sum_{i=0}^{l} n_i p_{2_i} | n_i, m_i \in \mathbb{N}, p_{1_i}, p_{2_i} \in \mathbb{N}^k \right\} \\
&= L(c_1 + c_2; p_{1_1}, \ldots, p_{1_j}, p_{2_1}, \ldots, p_{2_l}) \\
&= L(c_1 + c_2; P_1 \cup P_2)
\end{aligned}
$$

Example For $S_1 = \{c_1; p_1, p_2\} \cup \{c_2; p_3, p_4\}, S_2 = \{c_3; p_5, p_6\}$ we get $S = S_1 \otimes S_2 = \{c_1 + c_3; p_1, p_2, p_5, p_6\} \cup \{c_2 + c_3; p_3, p_4, p_5, p_6\}$.

The multiplication on the explicit semilinear set representation causes exponential growth in the size of the semilinear set. Let n be the number of linear sets in S_1 and m be the number of linear sets in S_2 then the number of linear sets in $S = S_1 \otimes S_2$ is polynomial in $n \cdot m$ because we calculate the Cartesian product of all linear sets of S_1 and S_2. Without simplification/minimization of the resulting semilinear set S, practical computation with these sets will be too expensive.

But in the corresponding automata A_1 and A_2 we encode binary representations of all vectors in S_1 and S_2. Therefore the individual linear sets are no longer accessible to us. But in this representation it is possible to calculate A for $\mathcal{L}(A) = S_1 \otimes S_2$ by adding up all accepting words from A_1 to all accepting words in A_2. This is done by creating an automaton which accepts $\{\vec{v_1} + \vec{v_2} | \vec{v_1} \in S_1, \vec{v_2} \in S_2\}$.

3.2.4. Kleene Star

If L is a formal language, then the Kleene closure L^* is defined as $\bigcup_{i \in \mathbb{N}} L^i$ where L^i is L concatenated i-times $\underbrace{LL \ldots L}_{i\text{-times}}$ [EP02]. In general the Kleene closure can be defined on semirings by $S^* = \sum_{i \in \mathbb{N}}^{\infty} S^i$[DKV09][Kui97]. However, we cannot practically sum up to infinity so we have to find another way to calculate S^*. In our case we are working with Parikh images (see section 2.3) so we can assume our semiring to be idempotent and commutative.

Semilinear sets can have many different representations and we will use one of them in this section for an example and the proof of our method to calculate the Kleene star. For example the explicit semilinear set representation can be converted easily to commutative regular expressions. We need a a mapping for

each component of our vectors in $\mathbb{N}^k$ to a letter v_k in the alphabet of the regular language. Then we can convert the semilinear set $S = L((1,0);(0,2)) \cup L((2,1);(1,0),(3,1))$ to the regular expression $v_1(v_2v_2)^* + v_1v_1v_2(v_1^* + (v_1v_1v_1v_2)^*)$ with $+$ being the alternative operator by writing the corresponding letter v_i for the i-th vector component m times if m is its value and for periods we star the corresponding expression. The resulting regular expression can also be easily transformed back so we can switch between these representations if one of them is better suited. Other symbolic representations are the NDDs, but in this case the transformation is not as efficient as with regular languages.

Example This is an example demonstrating the calculation of S^* in the domain of commutative regular expressions with $S = u_0u_1^* + v_0v_1^*$. We briefly recall that $(xy*)^* = 1 + xx^*y^*$ and $(x+y)^* = x^*y^*$ and $(x^*y^*)^* = x^*y^*$ [Koz94].

$$\begin{aligned} S^* &= (u_0u_1^* + v_0v_1^*)^* \\ &= (1 + u_0u_0^*u_1^*)(1 + v_0v_0^*v_1^*) \\ &= 1 + u_0u_0^*u_1^* + v_0v_0^*v_1^* + u_0v_0u_0^*u_1^*v_0^*v_1^* \end{aligned}$$

This is what we want to get as a result after applying the Kleene star to S. But this is not possible when working with NDDs representing semilinear sets because we do not have any way to work on the equations directly. We only have S and the constants (in our example u_0 and v_0).

We want to derive an algorithm with only addition and multiplication of S and the given constants because in the automata representation we do not have access to the linear sets anymore but only to S and the set of constants needed for S. It is possible to achieve parts of the missing factors by using powers of S.

$$\begin{aligned} S^2 &= (u_0u_1^* + v_0v_1^*)^2 \\ &= u_0^2u_1^* + v_0^2v_1^* + u_0v_0u_1^*v_1^* \end{aligned}$$

We see that we get a higher order constant (more constants of different factors combined in one) when using a higher power of S. Given a set S with 5 different constants, we have to calculate at least S^5 to get the constant $c_1c_2c_3c_4c_5$ for $c_i \in C = \text{constants}(S)$.

There are still periods missing in the result. In our example these missing periods are u_0^* and v_0^*, and the cannot be created by using powers of S. They can only be

obtained by multiplying S^i with $C^* = \prod_i c_i^*$ with $c_i \in C$. That way we can get u_0^* and v_0^* into the equation.

To get back to our example, we calculate $S^{*'} = 1 + S \cdot C^* + S^2 \cdot C^*$

$$\begin{aligned} S^{*'} &= 1 + (u_0 u_1^* + v_0 v_1^*)(u_0^* v_0^*) + \\ &\quad (u_0^2 u_1^* + v_0^2 v_1^* + u_0 v_0 u_1^* v_1^*)(u_0^* v_0^*) \\ &= 1 + u_0 u_1^* u_0^* v_0^* + v_0 v_1^* u_0^* v_0^* + \\ &\quad u_0^2 u_1^* u_0^* v_0^* + v_0^2 v_1^* u_0^* v_0^* + u_0 v_0 u_1^* v_1^* u_0^* v_0^* \\ &=^? \\ S^* &= 1 + u_0 u_0^* u_1^* + v_0 v_0^* v_1^* + u_0 v_0 u_0^* u_1^* v_0^* v_1^* \end{aligned}$$

We can see that $S^* \subseteq S^{*'}$ and $S^{*'} \subseteq S^*$ and therefore $S^* = S^{*'}$ when we look carefully at the equations. $S^{*'}$ has a longer and more complicated representation but both expressions represent the same set.

3.2.4.1. Proof

We claim that $S^* = 1 + (\sum_{k=1}^{n} S^k) \prod_{j=1}^{n} v_{i,0}^*$ and prove this by induction on the number of constants n in our semilinear set S. In this proof the constant of the i-th linear set is denoted with $v_{i,0}$ while the j-th period of the i-th linear set is denoted with $v_{i,j}$. For the limit of the products we use m_v or m_i, which represents the number of periods in the i-th linear set respectively the linear set which we mark with the letter v. For the base cases we omit the index i for the constants and periods which indicates the linear set and just use v_j.

The base case $n = 0$ is trivial with $S = 0$ and $S^* = 1 + 0 \cdot 0^* = 1 + 1 = 1$ (with idempotence). For $n = 1$ let $S = v_0 \cdot \prod_{j=1}^{n} v_j^*$. This results in $S^* = 1 + v_0 v_0^* \cdot \prod_{j=1}^{n} v_j^* = 1 + S^1 \cdot v_0^*$. The case $n = 2$ is more interesting and also brings an intuition for the operation. Let $S = v_0 \cdot \prod_{j=1}^{m_v} v_j^* + w_0 \cdot \prod_{j=1}^{m_w} w_j^*$ then we obtain with the equalities $(x + y)^* = x^* y^*$ and $(x^*)^* = x^*$:

$$
\begin{aligned}
S^* &= \left(v_0 \cdot \prod_{j=1}^{m_v} v_j^* + w_0 \cdot \prod_{j=1}^{m_w} w_j^* \right)^* \\
&= \left(1 + v_0 v_0^* \cdot \prod_{j=1}^{m_v} v_j^* \right) \left(1 + w_0 w_0^* \cdot \prod_{j=1}^{m_w} w_j^* \right) \\
&= 1 + v_0 v_0^* \cdot \prod_{j=1}^{m_v} v_j^* + w_0 w_0^* \cdot \prod_{j=1}^{m_w} w_j^* + v_0 w_0 v_0^* w_0^* \cdot \prod_{j=1}^{m_v} v_j^* \cdot \prod_{j=1}^{m_w} w_j^* \\
&=^? 1 + S^1 v_0^* w_0^* + S^2 v_0^* w_0^*
\end{aligned}
$$

It remains to show the last equality by showing that both equations include the other. The "$\subseteq$"-case is trivial and it can be seen easily that the first two sums are in $S^1 v_0^* w_0^*$ and the last one is included in $S^2 v_0^* w_0^*$. The other direction "$\supseteq$" requires a case distinction to show that all elements are included. For 1 we can trivially see that this is the case and proceed with $S^1 v_0^* w_0^*$, which is found in $v_0 v_0^* \cdot \prod_{j=1}^{m_v} v_j^*$ when we have the case that w_0^* is empty. If v_0^* is empty then it is included in $w_0 w_0^* \cdot \prod_{j=1}^{m_w} w_j^*$. If both v_0^* and w_0^* are present, we can find it in $v_0 w_0 v_0^* w_0^* \cdot \prod_{j=1}^{m_v} v_j^* \cdot \prod_{j=1}^{m_w} w_j^*$. The last case is $S^2 v_0^* w_0^*$, which can be easily found in $v_0 w_0 v_0^* w_0^* \cdot \prod_{j=1}^{m_v} v_j^* \cdot \prod_{j=1}^{m_w} w_j^*$.

Now we show that this also holds for $n+1$ and let $S = \sum_{i=1}^{n+1} v_{i,0} \cdot \prod_{j=1}^{m_i} v_{i,j}^* = \sum_{i=1}^{n} v_{i,0} \cdot \prod_{j=1}^{m_i} v_{i,j}^* + v_{n+1,0} \cdot \prod_{j=1}^{m_{n+1}} v_{n+1,j}^*$.

$$
\begin{aligned}
S^* &= \underbrace{\left(\sum_{i=1}^{n} v_{i,0} \cdot \prod_{j=1}^{m_i} v_{i,j}\right)^*}_{\text{induction hypothesis}} \cdot \left(v_{n+1,0} \cdot \prod_{j=1}^{m_{n+1}} v_{n+1,j}^*\right)^* \\
&= \left(1 + \left(\sum_{i=1}^{n} S^i\right) \cdot \prod_{j=1}^{n} v_{j,0}^*\right)\left(1 + v_{n+1,0}v_{n+1,0}^* \cdot \prod_{j=1}^{m_{n+1}} v_{n+1,j}^*\right) \\
&= 1 + \left(\sum_{i=1}^{n} S^i\right) \cdot \left(\prod_{j=1}^{n} v_{j,0}^*\right) + \left(v_{n+1,0}v_{n+1,0}^* \cdot \prod_{j=1}^{m_{n+1}} v_{n+1,j}^*\right) + \\
&\quad \left(\sum_{i=1}^{n} S^i\right) \cdot \left(\prod_{j=1}^{n} v_{j,0}^*\right) v_{n+1,0}v_{n+1,0}^* \cdot \prod_{j=1}^{m_{n+1}} v_{n+1,j}^* \\
&= 1 + \left(\sum_{i=1}^{n} S^i\right) \cdot \left(\prod_{j=1}^{n} v_{j,0}^*\right) + \left(v_{n+1,0}v_{n+1,0}^* \cdot \prod_{j=1}^{m_{n+1}} v_{n+1,j}^*\right) + \\
&\quad \left(\sum_{i=1}^{n} S^i\right) \cdot \left(\prod_{j=1}^{n+1} v_{j,0}^*\right) v_{n+1,0} \cdot \prod_{j=1}^{m_{n+1}} v_{n+1,j}^* \\
&=^? 1 + \left(\sum_{i=1}^{n+1} S^i\right) \cdot \left(\prod_{i=1}^{n+1} v_{i,0}^*\right) \\
&= 1 + \left(\sum_{i=1}^{n} S^i\right) \cdot \prod_{j=1}^{n+1} v_{j,0}^* + S^{n+1} \prod_{j=1}^{n+1} v_{j,0}^*
\end{aligned}
$$

The last equality has to hold and we start with showing the "$\subseteq$"-case:

- $1 \subseteq 1$ is trivial.
- $\left(\sum_{i=1}^{n} S^i\right) \cdot \left(\prod_{j=1}^{n} v_{j,0}^*\right)$ can be found in $\left(\sum_{i=1}^{n+1} S^i\right) \cdot \left(\prod_{i=1}^{n+1} v_{i,0}^*\right)$ with $v_{n+1,0}^* = \epsilon$.
- $\left(v_{n+1,0}v_{n+1,0}^* \cdot \prod_{j=1}^{m_{n+1}} v_{n+1,j}^*\right)$ is included in $S^1 \cdot v_{n+1,0}^* = \sum_{i=1}^{n+1} v_{i,0} \cdot \prod_{j=1}^{m_i} v_{i,j}^* \cdot v_{n+1,0}$, in the last iteration $n+1$ of the sum with the last period $v_{n+1,0}^*$.
- It remains to show that $\left(\sum_{i=1}^{n} S^i\right) \cdot \left(\prod_{j=1}^{n+1} v_{j,0}^*\right) v_{n+1,0} \cdot \prod_{j=1}^{m_{n+1}} v_{n+1,j}^*$ is included:
  - $\left(\sum_{i=1}^{n} S^i\right) \cdot \prod_{j=1}^{n+1} v_{j,0}^*$ if the periods of the last set/iteration $\prod_{j=1}^{m_{n+1}} v_{n+1,j}^*$ are missing.
  - $S^{n+1} \prod_{j=1}^{n+1} v_{j,0}^* \subseteq S^n \left(\prod_{j=1}^{n} v_{j,0}^*\right) \cdot S \cdot v_{n+1,0}^*$ otherwise

The other direction "$\supseteq$" can be shown by splitting up the sum and we see that

- $(\sum_{i=1}^{n} S^i) \cdot \prod_{j=1}^{n+1} v_{j,0}^*$ is included in
  - $(\sum_{i=1}^{n} S^i) \cdot (\prod_{j=1}^{n+1} v_{j,0}^*) v_{n+1,0} \cdot \prod_{j=1}^{m_{n+1}} v_{n+1,j}^*$ if there is at least one constant v_{n+1}
  - $(\sum_{i=1}^{n} S^i) \cdot (\prod_{j=1}^{n} v_{j,0}^*)$ otherwise.

The remaining $S^{n+1} \prod_{j=1}^{n+1} v_{j,0}^*$ can be rewritten to $S^n \cdot S \prod_{j=1}^{n+1} v_{j,0}^* = S^n \cdot (\sum_{i=0}^{n+1} v_{i,0} \cdot \prod_{j=1}^{m_i} v_{i,j}^*) \prod_{j=1}^{n+1} v_{j,0}^*$ which is included in:

- $(\sum_{i=1}^{n} S^i) \cdot (\prod_{j=1}^{n+1} v_{j,0}^*) v_{n+1,0} \cdot \prod_{j=1}^{m_{n+1}} v_{n+1,j}^*$ (last iteration n of the sum).

We have shown that both "$\subseteq$" and "$\supseteq$" holds and therefore our assumption $S^* = 1 + (\sum_{k=1}^{n} S^k) \prod_{j=1}^{n} v_{i,0}^*$ is true. □

3.2.4.2. Using a Minimal Basis Instead of Constants

The Kleene star is computable with the formula $S^* = 1 + (\sum_{k=1}^{n} S^k) \prod_{j=1}^{n} v_{i,0}^* = 1 + (\sum_{k=1}^{n} S^k) \cdot C^*$ where C^* is the product of all stared constants in n iterations for n constants (and therefore n linear sets). But as we are defining the Kleene star on NDDs we cannot directly extract the constants from an NDD efficiently. For this reason we would have to track all constants during all operations which is very inefficient as we face an exponential blow up during multiplication. In the Kleene star algorithm we only need the constants to generate the set C^*, but we do not need them for recreating the linear sets themselves. It would suffice to have a set B^* which is equivalent to C^*. The constants $v_{i,0}$ for $i \in [n]$ may be linear dependent and we could calculate a minimal basis B for the set which generates the same set as C^* when we apply the star operator to it.

One problem we now face is the missing number of constants respectively linear sets which are represented by the NDD, but it is needed for the upper bound of iterations in the algorithm. It is an open problem if the number of iterations $|B|$ in $S^* =^? 1 + (\sum_{k=1}^{|B|} S^k) \cdot B^*$ is a valid upper bound or if we loose precision. This problem can be circumvented by increasing the iterations until we reach the point where $(\sum_{k=1}^{m} S^k) = (\sum_{k=1}^{m+1} S^k)$.

3.2.5. Queries

3.2.5.1. Equality/Inclusion Tests

An interesting test which we want to be able to do with our sets is to test if two sets are equal or if one is included in the other. In the semilinear set representation we cannot simply check for equality because semilinear sets and Presburger formulae are missing the canonicity property. One can define sets which are equal but have a different representation and a check for equality would involve sophisticated tests to check if both sets include the other. On the other hand, it is simple to check if one automaton A is included in the other automaton B by checking if $A \cap \overline{B} = \varnothing$. There is an algorithm by Hopcroft which can test equality of two automata in $\mathcal{O}(|\Sigma| \cdot |A| \cdot |B|)$ [HK71]. But as we work with minimal DFAs we know that these DFA have a unique representation for a given set and we can simply check for equality by comparing if the states and transitions between the state are equal modulo the state ids.

3.2.5.2. Inclusion of a Vector in a Set

We want to check if a given vector is in the set. This is a hard problem in the explicit semilinear set representation as we have no direct way to find out which linear combination of the periods has to be used to create the vector in the general case.

Example Given the semilinear set $L = \{(1,4,2);(3,2,4),(2,3,2),(3,2,3)\} \cup \{(3,2,4);(2,0,1),(2,1,4)\}$ we cannot efficiently test if the vector $v = (17,18,19) \in L$. This is because $(17,18,19) = (1,4,2) + 1 \cdot (3,2,4) + 2 \cdot (2,3,2) + 3 \cdot (3,2,3)$, but to determine the factors $1,2,3$ for the periods we have to solve the NP-complete *knapsack problem*.

Still it is possible to represent the semilinear set L as a number decision diagram A. This automaton A recognizes words which represent binary representations of all solutions in L. Thus, we only have to ask A if it accepts the solution v in its binary representation, which can be done in $\mathcal{O}(|v|)$.

Part II.

Algorithms and Implementation

4. Algorithms

The algorithms described in this chapter are designed to work with automata by only using high-level operations like union, intersection and projection. For that reason some of the algorithms might seem overly complicated as we are not working directly on the transitions and states of the automata. In most cases the complexity of the algorithms would not change if we would use a more direct approach by working on the transitions and states.

4.1. Encoding of Automata Input

As described in section 2.4 we can recognize sets of numbers with a special kind of automata called *number decision diagrams*. In this thesis we also want to recognize vectors of natural numbers $\mathbb{N}^k$ so we have to encode these vectors of numbers in such a way that we are able to feed them into the automaton.

There are several ways to linearize the words which represent vectors of numbers. One would be to encode two words $a_0a_1 \ldots a_n$ and $b_0b_1 \ldots b_n$ by concatenating the input words $a_0a_1 \ldots a_nb_0b_1 \ldots b_n$.

Another possibility is to use an *interleaving* encoding like $a_0b_0a_1b_1 \ldots a_nb_n$. The example in figure 4.1 shows the interleaving LSBF encoding of the vector $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}_{10} = \begin{bmatrix} 10 \\ 01 \\ 11 \end{bmatrix}_2$ with sign bits. The first three transitions are the least significant bits of the three numbers. As the binary representation of 1 has only one bit, the sign bit of the first vector is already read in the fourth transition between the states 5 and 4. The loop at the end can read arbitrary many 0s for each of the vector components, but at least one 0 because of the sign bit. As we have three components, this loop has length three.

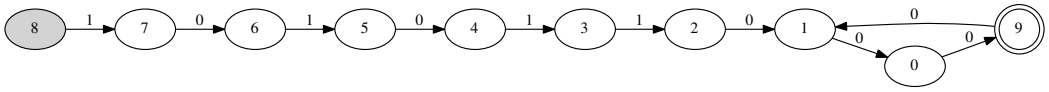


Figure 4.1.: Automaton accepting $(1, 2, 3)$ with interleaving encoding as $101011(000)^*$

4.2. Construction of an Automaton for the Kleene Star

As described in section 3.2.4 we can calculate the Kleene star closure by adding up increasing powers of the original set multiplied with the constants c of S used as periods p until we reach the fixed point. If we would know the number of linear sets in the semilinear set, we would have an upper bound of n in the calculation of $S^* = 1 + (\sum_{k=1}^{n} S^k) \prod_{j=1}^{n} v_{i,0}^*$. But as we want to calculate the Kleene star with a minimal basis of the constants we will iterate the loop until our resulting set does not change anymore.

Input: Automaton A
Output: Kleene star closure on the recognized set S
$B \leftarrow$get minimal basis of constants of A;
$B_* \leftarrow$constants*;
$S_\Sigma \leftarrow$1;
`/* semiring 1 */`
$S_{Acc} \leftarrow$1;
`/* init with semiring 1 */`
result $\leftarrow$1;
repeat
  old_result $\leftarrow$result;
  $S_{Acc} \leftarrow S_{Acc} \cdot A$;
  $S_\Sigma \leftarrow S_\Sigma + S_{Acc}$;
  result = $1 + B_* \cdot S_\Sigma$;
until *old_result* $\equiv$ *result*;
return *result*;

Algorithm 1: Calculating the Kleene star of S recognized by A

Algorithm 1 uses the minimal basis B of the constants from the semilinear set S represented by A and converts them to B^*. We use the fact, described in section 3.2.4, that we do not need the actual set of stared constants C^*, but it suffices to have an equivalent set $B^* = C^*$ which we use to accelerate the calculation of the Kleene star. This minimal basis is stored explicitly with the automaton and is always minimized after each operation like multiplication or addition to elimi-

nate linear dependent vectors. These vectors of the minimal basis of constants are converted to periods by creating linear equations which are used to compute the automaton to recognize these new periods.

The sum $\sum_{i=1}^{k} S^i$ is calculated by using Horner's method such that we can save many expensive multiplication steps needed for calculating S^i. The resulting sum is multiplied with the set of periods B_* and added to the semiring 1 which is $1 = (\underbrace{(0, \ldots, 0)}_{k-\text{times}}; \varnothing)$.

4.3. Sum Automaton

To multiply semilinear sets in automata representation we need to construct an automaton which recognizes the set $\{(x, y, z) \in \mathbb{N}^3 | x + y = z\}$ to be able to add the vectors of two automata. We assume our input automaton accepts input $\vec{x} \in \mathbb{N}^3$ such that we have three components with x being the first operand, y the second operand and z is the solution. Then, we rewrite the formula $x + y = z$ to $1 \cdot x + 1 \cdot y - 1 \cdot z = 0$ and get the coefficients $(1, 1, -1)$ for the variables (x, y, z).

For the actual construction of this automaton with GENEPI we can use the function `genepi_set_linear_equality()` which takes an array of coefficients, in our case $(1, 1, -1)$ and a constant 0. GENEPI then returns the automaton shown in figure 4.2. Section 3.2.1.1 describes in detail how such an automaton is constructed.

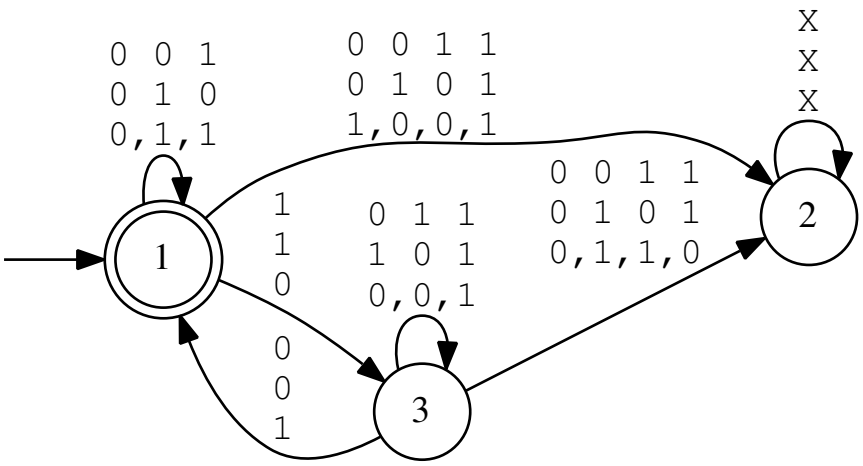


Figure 4.2.: Automaton accepting $\{(x, y, z) \in \mathbb{N}^3 | x + y = z\}$

The sum automaton in figure 4.2 accepts all triples where the third component is the sum of the first two components. If we are in state 1 in the automaton then we can use the self loop with the three vectors $(0,0,0)^T, (0,1,1)^T, (1,0,1)^T$ and stay in state 1. If the first two components are 1 then component 3 has to be 0 and we have to remember a *carry bit*. This is done by going into state 3. We stay in state 3 as long as we have to remember the carry bit. With the vector $(0,0,1)^T$ we can leave the carry state and return to the accepting state 1 again. All transitions to the invalid state 2 are invalid words which are not recognized by the automaton.

4.4. Construction of Automata Recognizing a Vector

To create an automaton which recognizes a specific vector $v \in \mathbb{N}^k$ we can can use the GENEPI function `genepi_set_linear_equality()` to accept the solutions of a set of linear equations. We explained in section 3.2.1, how we can transform a vector to a system of linear equations which can then be transformed to an NDD.

In our algorithm 2 we create for every component i of $v \in \mathbb{N}^k$ an automaton A_i which recognizes the i-th component of the vector. Next we split the system of linear equations $\bigwedge_{i=1}^{k} 1 \cdot x_i = v_i$ into k independent equations. The coefficients α_i of the linear equation for each component are $\alpha_{j\in[k];j\neq i} = 0; \alpha_i = 1; c = v_i$, which we can pass to the GENEPI function `genepi_set_linear_equality()` and get A_i in return. These automata A_i are then intersected with the set N that recognizes $\mathbb{N}^k$. The result of our algorithm is $A = N \cap A_1 \cap \ldots \cap A_k$.

This construction looks inefficient as we might create the automaton in a more direct way. For example to read the vector $\begin{bmatrix} 100 \\ 011 \end{bmatrix}$ we could simply generate the automaton by creating the transitions directly from the given vector where we always take the last bit of each component and create the transition. This would produce the automaton $(p) \xrightarrow{\begin{bmatrix} 0 \\ 1 \end{bmatrix}} (p') \xrightarrow{\begin{bmatrix} 0 \\ 1 \end{bmatrix}} (p'') \xrightarrow{\begin{bmatrix} 1 \\ 0 \end{bmatrix}} (p''') \in F$. But we want to restrict our algorithms to high-level functions to be independent of the underlying data structure.

4.5. Construction of Automata Recognizing a Period

The creation of an automaton recognizing a period $\mathbb{N} \cdot g$ for $g \in \mathbb{N}^k$, which we see in algorithm 3, works very similar to the creation of an automaton recog-

Input: Vector $\vec{v} \in \mathbb{N}^k$
Output: Automaton A which accepts the representation of $\vec{v}$
Result ←CreateAutomatonAccepting$\mathbb{N}^k$;
for $i = 0$ **to** k **do**
    A_i ←FromLinearEquation($v_i = x_i$);
    Result ←Result $\cap A_i$;
end
return *Result*

Algorithm 2: Calculating automaton accepting a constant

nizing a vector. We already described the mathematical background in section 3.2.1, where we noted that we introduce a new variable μ in our system of equations. The used system of equations is $\bigwedge_{i=1}^{k} -x_i + g_i \cdot \mu = 0$ and we pass them to `genepi_set_linear_equality()`. This is not the final automaton as it has an additional μ-component, which we would have to feed to the automaton in order to check a word for acceptance. But we are only interested in the existence of such a μ. In section 3.2.1 we applied the existential quantifier to the corresponding formula to get solutions for all μs. In the automaton representation, we can project the μ component and get the final automaton.

Input: Vector $\vec{g} \in \mathbb{N}^k$
Output: Automaton A which accepts the representation of the period $\vec{g}$
Result ←CreateAutomatonAccepting$\mathbb{N}^k$;
for $i = 0$ **to** k **do**
    A_i ←FromLinearEquation($-1 \cdot x_i + g_i \cdot \mu = 0$);
    Result ←Result $\cap A_i$;
end
Result ←π_μ(Result);
return *Result*

Algorithm 3: Calculating automaton accepting a period

4.6. Multiplication on Automata – High Level

To implement multiplication on automata we can only rely on automata operations like intersection and projection. In section 3.2.3 we describe how multiplication is defined on semilinear sets, but this definition cannot be applied directly to automata. The definition $S = S_1 \otimes S_2 = \{L_1 \otimes L_2 | L_1 \in S_1, L_2 \in S_2\}$ would require us to be able to access the linear sets in S, which is not possible. Instead,

we can use an alternative definition $S = \{\vec{v_1} + \vec{v_2} | v_1 \in S_1, v_2 \in S_2\}$. Multiplication of two semilinear sets is the addition of each vector of the first operand with each vector of the second operand. This is possible as we have all vectors respectively all binary representations of them in the automaton. We can do this by adding up all included vectors of both automata with the sum automaton A_Σ we defined in section 4.3, which accepts $\{(x, y, z) \in \mathbb{N}^3 | x + y = z\}$.

4.6.1. Automata with One Variable

We start by assuming all vectors $\vec{v} \in \mathbb{N}^1$ to be of dimension 1. Let A_1 and A_2 be our input automata then we like to create the automaton A which accepts as its language the binary representation of the set $\{v | v_1 + v_2 = v, v_1 \in A_1, v_2 \in A_2\}$. We already have the sum automaton A_Σ that accepts $\{(x, y, z) \in \mathbb{N}^3 | x + y = z\}$, which calculates the sum of the first two components and returns the summand in the third component. But for this approach the two operands have to be in the same automaton.

To combine the two operands in one automaton we have to change the encoding of the input automata. We create two new automata A'_1 and A'_2 which have three components each. The encoding of A'_1 will be $(a_1, \mathbb{N}, \mathbb{N})$, which accepts a vector with the original language of A_1 in its first component and all natural numbers in the remaining two components. A'_2 has a similar encoding but with the original language in the second component $(\mathbb{N}, a_2, \mathbb{N})$. This operation is called *inverse projection* and defined as $\pi_s(A) = \{(x_1, \ldots, x_n) | \exists v \in A, v[1] = x_{i_1} \wedge \ldots \wedge v[n] = x_{i_n}\}$, where s is a bit pattern like $(1, 1, 0, 1, 1, 0)$ and i_j is the j-th component of s which is 0. In our example this results in $i_1 = 3$ and $i_2 = 6$. It takes an automaton A and changes the encoding in such a way that the i-th component of A will be at position i_j and each position l with $s[l] = 1$ accepts all natural numbers $\mathbb{N}$. The dimension of the new automaton A' is equal to the length of the bit pattern s. We use a short notation $\pi^{-1}_{[x,y,z]_n}(A)$, which implies a bit pattern of length n and the bits x, y and z are set to 0 and all others are set to 1. The amount of 0 in the bit pattern has to match the dimension of A.

With this operation we can define $A'_1 = \pi^{-1}_{[1]_n}(A_1)$ and $A'_2 = \pi^{-1}_{[2]_n}(A_2)$ with n is the dimension of vectors in A_i times 3. These two automata are then intersected: $A' = A'_1 \cap A'_2$. The language of A' is the set of the binary representations of A_1 and A_2 in the first two components $\{(a_1, a_2, \mathbb{N}) | a_1 \in A_1, a_2 \in A_2\}$. Now we can intersect the automaton A' with the sum automaton A_Σ to get an intermediate result $A_I = A' \cap A_\Sigma = A_1 \cap A_2 \cap A_\Sigma$. A_I accepts all binary representations of the set $\{(a_1, a_2, a) | a_1 + a_2 = a, a_1 \in A_1, a_2 \in A_2\}$. The final automaton A should

only contain the summand so we project the first two components and get $A = \pi_{\{1,2\}}(A_I)$, which accepts $\{a | a_1 + a_2 = a, a_1 \in A_1, a_2 \in A_2\}$.

4.6.2. Automata with Many Variables

We just described the algorithm for automata which have only one variable/vectors $\vec{v} \in \mathbb{N}^1$ of dimension one. But the algorithm should also handle input automata that accept vectors $\vec{v} \in \mathbb{N}^k$. To extend the algorithm we have to modify the encoding of A_1' and A_2' to include all components of A_1 and A_2. As we use the interleaving encoding we define $A_1' = \pi^{-1}_{[3i-2|i\in[k]]_n}(A_1)$, with n is the dimension of vectors in A_i times $3k$. Thus, the encoding of A is transformed from $(a_1, a_2, \ldots, a_k)$ to $(a_1, \mathbb{N}, \mathbb{N}, a_2, \mathbb{N}, \mathbb{N}, \ldots, a_k, \mathbb{N}, \mathbb{N})$. Similarly we define $A_2' = \pi^{-1}_{[3i-1|i\in[k]]_n}(A_2)$ and transform $(b_1, b_2, \ldots, b_k)$ to $(\mathbb{N}, b_1, \mathbb{N}, \mathbb{N}, b_2, \mathbb{N}, \ldots, \mathbb{N}, b_k, \mathbb{N})$. Then we proceed as before and intersect the transformed automata to get $A' = A_1' \cap A_2' = \pi^{-1}_{[3i-2|i\in[k]]_n}(A_1)$ $\cap A_2' = \pi^{-1}_{[3i-1|i\in[k]]_n}(A_2)$, which has the encoding $(a_1, b_1, \mathbb{N}, a_2, b_2, \mathbb{N}, \ldots, a_k, b_k, \mathbb{N})$.

From here there are two possible ways of computing the sum of the respective sums. The first way is to create a large sum automaton $A_{\Sigma_{[k]}}$ which calculates the sum of all a_i and b_i components at once and intersect this sum automaton $A_{\Sigma_{[k]}}$ with A'. But this option gets problematic as $A_{\Sigma_{[k]}}$ tends to get very big for growing k. An analysis of this problem is found in section 5.2.2.

One way to circumvent this big sum automaton is to use the generic sum automaton A_Σ which calculates one component sum and apply it to all a_i, b_i sequentially. As the individual resulting sum $c_i = a_i + b_i$ depends only on the individual a_i and b_i this does not change the result of our algorithm. We can even permute the order in which we process the components. To operate on each of the components we have to inverse project the components from the generic sum automaton to the respective component to be calculated next. For example to calculate the second component $c_2 = a_2 + b_2$ we calculate the sum automaton $A_{\Sigma_2} = \pi^{-1}_{[4,5,6]_{3n}}(A_\Sigma)$ by positioning the generic sum automaton to the position of the second components. The resulting automaton for input vectors $\vec{v} \in \mathbb{N}^2$ can be seen in figure 4.3.

The intermediate automaton $A_I = A' \cap A_{\Sigma_1} \cap \ldots \cap A_{\Sigma_k}$ is created by intersecting all these small sum automata A_{Σ_i} with the combined input automaton A' we created earlier. A_I recognizes the set $\{(a_1, b_1, c_1, \ldots, a_k, b_k, c_k) : a_i \in A_1, b_i \in A_2, c_i = a_i + b_i, i \in [k]\}$. To get the final result automaton we project all a_i and b_i components $A = \pi_{\{3i-2|i\in[k]\}\cup\{3i-1|i\in[k]\}}(A_I)$ to recognize the set $\{(c_1, \ldots, c_k) : a_i \in A_1, b_i \in A_2, c_i = a_i + b_i, i \in [k]\}$.

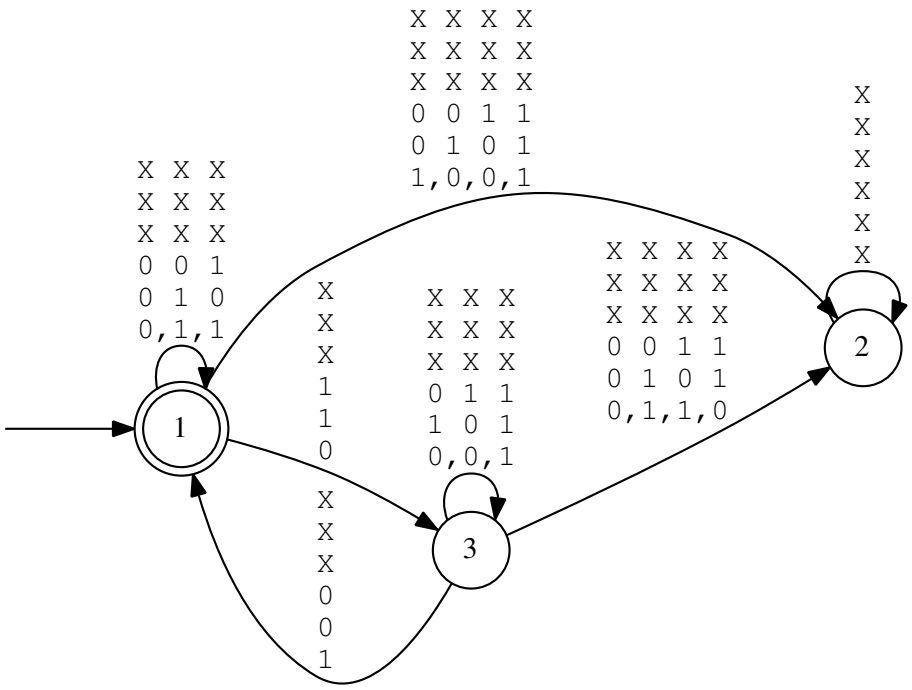


Figure 4.3.: Sum automaton A_{Σ_2} which calculates the sum of a_2 and b_2 for input vectors $\vec{v} \in \mathbb{N}^2$

4.7. Direct Construction of the Multiplication Automaton

Given the two input automata $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ und $A_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ we can also directly construct the multiplication automaton $A = (Q_1 \times Q_2 \times \{0,1\}, \Sigma, \delta, q_1 \times q_2 \times 0, F_1 \times F_2 \times 0)$ with the rules:

i) $\delta_1(q_1, 0) = q_1' \wedge \delta_2(q_2, 0) = q_2' \Rightarrow \delta((q_1, q_2, 0), 0) = (q_1', q_2', 0)$

ii) $\delta_1(q_1, 0) = q_1' \wedge \delta_2(q_2, 1) = q_2' \Rightarrow \delta((q_1, q_2, 0), 1) = (q_1', q_2', 0)$

iii) $\delta_1(q_1, 1) = q_1' \wedge \delta_2(q_2, 0) = q_2' \Rightarrow \delta((q_1, q_2, 0), 1) = (q_1', q_2', 0)$

iv) $\delta_1(q_1, 1) = q_1' \wedge \delta_2(q_2, 1) = q_2' \Rightarrow \delta((q_1, q_2, 0), 0) = (q_1', q_2', 1)$

v) $\delta_1(q_1, 0) = q_1' \wedge \delta_2(q_2, 0) = q_2' \Rightarrow \delta((q_1, q_2, 1), 1) = (q_1', q_2', 0)$

vi) $\delta_1(q_1, 0) = q_1' \wedge \delta_2(q_2, 1) = q_2' \Rightarrow \delta((q_1, q_2, 1), 0) = (q_1', q_2', 1)$

vii) $\delta_1(q_1, 1) = q_1' \wedge \delta_2(q_2, 0) = q_2' \Rightarrow \delta((q_1, q_2, 1), 0) = (q_1', q_2', 1)$

viii) $\delta_1(q_1, 1) = q_1' \wedge \delta_2(q_2, 1) = q_2' \Rightarrow \delta((q_1, q_2, 1), 1) = (q_1', q_2', 1)$

Input: Automaton A_1, Automaton A_2
Output: Automaton A which contains the result of $A_1 \otimes A_2$
$A'_1 \leftarrow \pi^{-1}_{3i,i\in[k]}(A_1)$;
$A'_2 \leftarrow \pi^{-1}_{3i+1,i\in[k]}(A_2)$;
$A_\Sigma \leftarrow$sum_automaton;
$A' \leftarrow A'_1 \cap A'_2$;
Result $\leftarrow$CreateAutomatonAccepting$\mathbb{N}^k$;
for $i = 1$ **to** k **do**
  $A_{\Sigma_i} \leftarrow \pi^{-1}_{\{3i-2,3i-1,3i\}}(A_\Sigma)$;
  Result $\leftarrow$Result $\cap A_{\Sigma_i}$;
end
Result = $\pi_{\{3i-2|i\in[k]\}\cup\{3i-1|i\in[k]\},i\in[k]}$(Result);
return A

Algorithm 4: Calculating the multiplication automaton via intersection with A_Σ

The third element of the tuple in the resulting automaton indicates the carry bit. The transition function can be read intuitively from the generic sum automaton shown in section 4.3. With this construction the sum is implicitly calculated during the construction and not in the intersection of the three automata. We do not have to calculate the big intermediate intersection automata and only create the resulting non-deterministic automaton, which we can determinize and minimize to get the final NDD.

The resulting sum automaton A is a product automaton of A_1 and A_2, where each product state exists with state 0 and state 1.

We then create transitions in the result automata such that they accept the sum of the numbers represented by the words of A_1 and A_2. The state 0 and state 1 will be used as a carry state which represents that we have a carry bit at the moment.

We start at $q_1 \times q_2 \times 0$. If we have a transition $q_1 \xrightarrow{0} q'_1$ in A_1 and a transition $q_2 \xrightarrow{0} q'_2$ in A_2, we add a transition $q_1 \times q_2 \times 0 \xrightarrow{0} q'_1 \times q'_2 \times 0$ in A. With doing this we calculate the sum of $0 + 0 = 0$. There are two other possibilities which leave us with no carry bit i.e. $q_1 \xrightarrow{0} q'_1 \wedge q_2 \xrightarrow{1} q'_2$ and $q_1 \xrightarrow{1} q'_1 \wedge q_2 \xrightarrow{0} q'_2$. They represent $0+1 = 1$ resp. $1+0 = 1$ so we have to add the transition $q_1 \times q_2 \times 0 \xrightarrow{1} q'_1 \times q'_2 \times 0$.

If we have the transitions $q_1 \xrightarrow{1} q'_1 \wedge q_2 \xrightarrow{1} q'_2$ in the original automaton, this represents $1 + 1 = 10$, where we are left with a carry bit. To remember this carry bit we use the states $q_{1,i} \times q_{2,i} \times \mathbf{1}$. So we add the transition $q_1 \times q_2 \times 0 \xrightarrow{0} q'_1 \times q'_2 \times 1$ to the automaton.

A state is a final/accepting state if both q_1 and q_2 in A_1 resp. A_2 are accepting states.

These have been all possible transitions starting at $q_1 \times q_2 \times \mathbf{0}$. Now we handle the cases where we are in $q_1 \times q_2 \times \mathbf{1}$ (we now have a carry bit). We have again three possible cases were we still have a carry bit and therefore stay in the carry states. These are the input transitions $q_1 \xrightarrow{0} q_1' \wedge q_2 \xrightarrow{1} q_2'$ and $q_1 \xrightarrow{1} q_1' \wedge q_2 \xrightarrow{0} q_2'$, which represent $0 + 1(+1) = 10$ resp. $1 + 0(+1) = 10$. For these we add the transition $q_1 \times q_2 \times 1 \xrightarrow{0} q_1' \times q_2' \times 1$ and $q_1 \xrightarrow{1} q_1' \wedge q_2 \xrightarrow{1} q_2'$ which represents $1 + 1(+1) = 11$ for which we add the transition $q_1 \times q_2 \times 1 \xrightarrow{1} q_1' \times q_2' \times 1$.

States where we carry a bit cannot be accepting states. The only way to reach a final state from $q_1 \times q_2 \times \mathbf{1}$ is to have the transitions $q_1 \xrightarrow{0} q_1' \wedge q_2 \xrightarrow{0} q_2'$, which represent $0+0(+1) = 1$. We add $q_1 \times q_2 \times 1 \xrightarrow{1} q_1' \times q_2' \times \mathbf{0}$. If q_1' and q_2' are accepting states, $q_1' \times q_2' \times 0$ is also an accepting state. This is because if we have accepted a word in A_1 and A_2 and do not carry a carry bit, we have simultaneously read two accepting words from the input automata and implicitly calculated the sum in the resulting sum automaton.

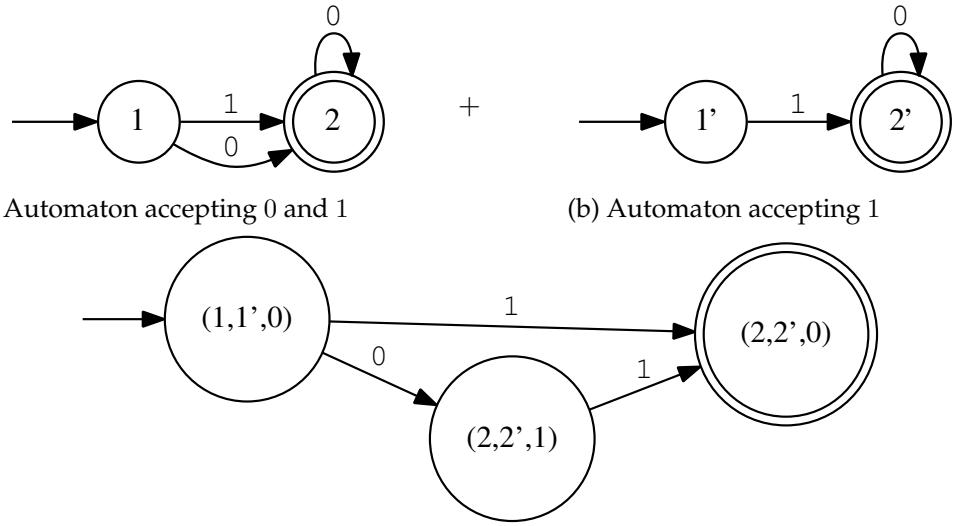
An example for this construction can be seen in figure 4.4, where the sum of two automata is calculated. The first automaton accepts 0 and 1 while the second one only accepts 1. When using the transition rules from above we construct an automaton which can read 1 and 2 in binary representation with LSBF encoding.

4.7.1. Proof

In this section we will use the notation $\langle x \rangle$ to denote the value of the word x, which is a binary representation in LSBF encoding. Additionally $[x]$ is used for the word representation of the value x. To show that our construction correctly calculates the sum automaton we have to prove two statements. First, that for every accepted word $[x] \in \mathcal{L}(A_1)$ and $[y] \in \mathcal{L}(A_2)$ we have an accepting word $[z] \in \mathcal{L}(A_{1+2})$ with $z = x + y$. And secondly, that every $z \in A_2$ can be divided into x and y such that they are accepted in $\mathcal{L}(A_1)$ respectively $\mathcal{L}(A_2)$.

"⇒": To show the first part we assume that we have a word x that is accepted in A_1 starting at state p, and accepting in state p', $(p) \xrightarrow{x} (p') \in F_1$ and a word y such that $(q) \xrightarrow{y} (q') \in F_2$ with states q, q' in A_2. Then, we have to show two cases:

a) $(p, q, 0) \xrightarrow{z=[\langle x \rangle + \langle y \rangle]} (p', q', 0) \in F_{1+2}$



(a) Automaton accepting 0 and 1

(b) Automaton accepting 1

(c) Sum automaton accepting 1 and 2 as the sum of a) and b)

Figure 4.4.: Example of a sum of two automata

b) $(p, q, 1) \xrightarrow{z=[\langle x\rangle+\langle y\rangle+1]} (p', q', 0) \in F_{1+2}$

Induction over $n = \min\{|x|, |y|\}$ is used to prove this. We start the induction at $n = 0$ and know that $|x| = 0 \vee |y| = 0$ and without loss of generality we can assume $|x| = 0$, $x = \epsilon$. Because of $x = \epsilon$ we know that $p = p'$ and therefore $p \in F_1$ and with $(q) \xrightarrow{y} (q')$ and $q' \in F_2$ we know by construction of A_{1+2} that $(p, q', 0) \in F_{1+2}$.

The word z is accepted in both cases:

1) $(p, q, 0) \xrightarrow{z=y} (p, q', 0) \in F_{1+2}$ by rule i) and ii) as we always read 0 in A_1.

2) $(p, q, 1) \xrightarrow{z=[\langle y\rangle+1]} (p', q', 0) \in F_{1+2}$

In case 2) we use induction over $m = |y|$. For $m = 0$ and $y = \epsilon$ we have $p = p' \wedge q = q'$ and read 1, and it is trivial to see that $(p, q, 1) \xrightarrow{1} (p, q, 0) \in F_{1+2}$ with transition rule iv). Now we handle the induction step $m \to m + 1$ to read $[\langle y\rangle + 1]$ and use a case distinction on y_0 where $y = y_0y'$ such that y' is a shorter word than y:

- $y_0 = 0$: $z = [1]y' \Rightarrow (p, q, 1) \xrightarrow{1} (p', q', 0) \xrightarrow{y'} (p'', q'', 0)$ by rule v) and the induction hypothesis for the sequence which reads y'.

- $y_0 = 1$: $z = [0]y' \Rightarrow (p, q, 1) \xrightarrow{0} (p', q', 1) \xrightarrow{[\langle y' \rangle + 1]} (p'', q'', 0)$ due to the transition rule vi) and the induction hypothesis.

In the induction step $n \to n + 1$ we have the words $x = x_0x'$ and $y = y_0y'$ and do a case distinction on the values of x_0 and y_0.

- $x_0 = 0 \vee y_0 = 0$ (three cases) $z = [\langle x \rangle + \langle y \rangle] = [\langle x_0 \rangle + \langle y_1 \rangle][\langle x' \rangle + \langle y' \rangle]$ with the sequence $(p, q, 0) \xrightarrow{[\langle x_0 \rangle + \langle y_1 \rangle]} (p', q', 0) \xrightarrow{[\langle x' \rangle + \langle y' \rangle]} (p'', q'', 0)$ where we use the first three transition rules for the first transition and the induction hypothesis on the sequence for the shorter word $[\langle x' \rangle + \langle y' \rangle]$.
- $x_0 = 1 \wedge y_0 = 1 \Rightarrow z = [\langle x \rangle + \langle y \rangle] = [0][\langle x' \rangle + \langle y' \rangle + 1]$ has to be read with a sequence $(p, q, 0) \xrightarrow{[0]} (p', q', 0) \xrightarrow{[\langle x' \rangle + \langle y' \rangle + 1]} (p'', q'', 0)$ where we can apply the induction hypothesis on the smaller word.

"$\Leftarrow$": The other direction, that all words accepted by A_{1+2} have to be the sum of an $x \in \mathcal{L}(A_1)$ and a $y \in \mathcal{L}(A_2)$, can be shown by splitting $z = [\langle x \rangle + \langle y \rangle]$, which is accepted by $(p, q, 0) \xrightarrow{z} (p', q', 0)$, into its two components x and y. The first component x, which has to be accepted by A_1 by the sequence $(p) \xrightarrow{x} (p') \in F(A_1)$ and the second component y in A_2, which is read with $(q) \xrightarrow{y} (q')$. From the construction we see that we can extract both summands by looking at the sequence of states that are accepting z, and reconstruct the word x by looking at the first component p of the state tuple $(p, q, \{0, 1\})$ and use the transitions between these states in A_1 to extract it. The same can be done for the word y and the second component q to reconstruct the sequence of states in A_2 which have been used to accept y. This might not be a unique word because if both transitions $(p) \xrightarrow{0} (p') \wedge (p) \xrightarrow{1} (p')$ are in A_1, then both words would be a candidate for the summand.

To prove this direction we have to show that:

a) $(p, q, 0) \xrightarrow{z=[\langle x \rangle + \langle y \rangle]} (p', q', 0') \in F_{1+2} \Rightarrow (p) \xrightarrow{x} (p') \in F_1 \wedge (q) \xrightarrow{y} (q') \in F_2$

b) $(p, q, 1) \xrightarrow{z=[\langle x \rangle + \langle y \rangle + 1]} (p', q', 0) \in F_{1+2} \Rightarrow (p) \xrightarrow{x} (p') \in F_1 \wedge (q) \xrightarrow{y} (q') \in F_2$.

For this we use an induction over $n = |z|$ and start with $n = 0, z = \epsilon$. For $z = \epsilon$ we have $p = p' \wedge q = q'$ and show both a) and b) with

1) $(p, q, 0) \xrightarrow{0} (p, q, 0) \in F_{1+2}$ then by construction $\epsilon \in \mathcal{L}(A_1) \Rightarrow p \in F_1 \wedge \epsilon \in \mathcal{L}(A_2) \Rightarrow q \in F_2$.

2) $(p,q,1) \xrightarrow{1} (p,q,0) \in F_{1+2}$ then this transition was added by the transition rule v) and we have $(p) \xrightarrow{0} (p) \wedge (q) \xrightarrow{0} (q)$ and by construction of $(p,q,0) \in F_{1+2}$ we have $p \in F_1 \wedge q \in F_2$.

In the induction step $n \to n+1$ we write $[\langle x \rangle + \langle y \rangle] = z = z_0 z'$ such that z_0 is the last bit and z' is a smaller word which can already be read. We will do a case distinction on z_0 and for $z_0 = 0$ we have to show:

1) $(p,q,0) \xrightarrow{0} (p',q',0) \xrightarrow{[\langle x' \rangle + \langle y' \rangle]} (p'',q'',0) \in F_{1+2}$, which is achieved with the first transition rule with $x_0 = 0 \wedge y_0 = 0$ and the induction hypothesis.

2) $(p,q,1) \xrightarrow{0} (p',q',0) \xrightarrow{[\langle x' \rangle + \langle y' \rangle + 1]} (p'',q'',0) \in F_{1+2}$

Part 2) consists of two cases because either we read a 1 in A_1 or in A_2 while the other reads a 0:

- $(p) \xrightarrow{0} (p') \xrightarrow{x'} (p'') \in F_1 \wedge (q) \xrightarrow{1} (q') \xrightarrow{y'} (q'') \in F_2$ by rule vi) and induction hypothesis for x' and y'.
- $(p) \xrightarrow{1} (p') \xrightarrow{x'} (p'') \in F_1 \wedge (q) \xrightarrow{0} (q') \xrightarrow{y'} (q'') \in F_2$ by rule vii) and induction hypothesis.

The second case is $z_0 = 1$ for which we show both a) and b) with:

1) $(p,q,0) \xrightarrow{[1]} (p',q',0) \xrightarrow{[\langle x' \rangle + \langle y' \rangle]} (p'',q'',0) \in F_{1+2}$ which has analogous to 2) in $z_0 = 0$ the two cases, that we read either a 1 in A_1 and 0 in A_2 or we read 0 in A_1 and 1 in A_2.

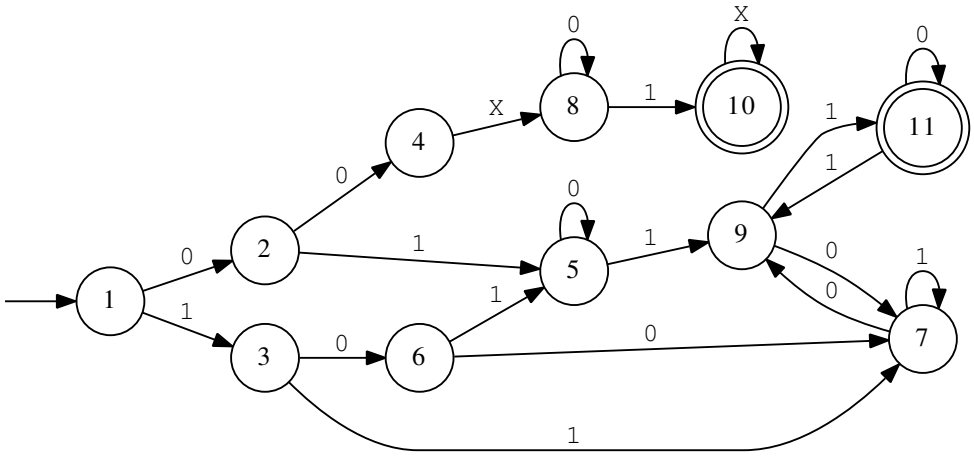
2) This is analogous to 1) in the case $z_0 = 0$ where we read $(p,q,1) \xrightarrow{1} (p',q',1) \xrightarrow{[\langle x \rangle + \langle y \rangle + 1]} (p'',q'',0) \in F_{1+2}$. This transition sequence can be added by the last transition rule and for the rest we can apply the induction hypothesis.

This concludes the proof. We now know that the sum automaton recognizes all sums of all words that are accepted by A_1 and A_2 and all words recognized by A_{1+2} are sums of words accepted by A_1 and A_2. □

4.8. Finding Constants in an Automaton

There have been several attempts (see section 2.5.3 and 1.2) to recover the semilinear set from an automaton. In section 3.2.4 and 4.2 we stated that we need only the constants of the semilinear set for the construction of the Kleene star automaton and we can neglect the periods. This section describes an algorithm, which is a

good heuristic to find constants by operating on the directed graph which represents our automaton. We then find all *simple paths* from the start state to all accepting end states. A *simple path* is a path where each vertex is included only once. We suppose that we can find a superset of necessary constants that are needed to recreate the semilinear set if we would also find the periods, but have not been able to prove it yet.



- $1 \xrightarrow{0} 2 \xrightarrow{0} 4 \xrightarrow{0} 8 \xrightarrow{1} 10 = 8$
- $1 \xrightarrow{0} 2 \xrightarrow{0} 4 \xrightarrow{1} 8 \xrightarrow{1} 10 = 12$
- $1 \xrightarrow{0} 2 \xrightarrow{1} 5 \xrightarrow{1} 9 \xrightarrow{1} 11 = 14$
- $1 \xrightarrow{1} 3 \xrightarrow{0} 6 \xrightarrow{1} 5 \xrightarrow{1} 9 \xrightarrow{1} 11 = 29$
- $1 \xrightarrow{1} 3 \xrightarrow{0} 6 \xrightarrow{0} 7 \xrightarrow{0} 9 \xrightarrow{1} 11 = 17$
- $1 \xrightarrow{1} 3 \xrightarrow{1} 7 \xrightarrow{0} 9 \xrightarrow{1} 11 = 11$

Figure 4.5.: All simple paths and the corresponding constants in an example automaton accepting $S = L_1(8; 4) \cup L_2(8; 3)$

Example In figure 4.5 one can see all found simple paths in the automaton from the start state 1 to both accepting states 10 and 11. The result is a superset of constants as we only need the constants 8 for both our linear sets L_1 and L_2. Note that we do not always get the same constants which have been in the input because the result might show that the canonical form does not need this constant. For example see the set $S_2 = L_3(4; 2) \cup L_4(10; 3)$ which is equivalent to the semilinear

set $S_2 \equiv S_3 = L_5(4;2) \cup L_6(13;3)$. We can see that the constant 10 of L_4 is also included in L_3 because $10 = 4 + 3 \cdot 2$. So we do not need 10 in L_6 and can safely use 13 as constant.

Note that we require all simple paths from the start state to all end states, not only the shortest ones. We therefore modify a depth-first search approach to find cycle free paths.

The idea is to begin at the starting node and descend into the first neighbour state. We save the already visited states in a sequence called $visited$ to be able to check if we already visited a state. We do a depth-first descend and always descend into the first neighbour of the new state until we find an accepting end state. In this situation we rebuild our path by looking at $visited$ and save the sequence of matching words along this path. As there might be more than one transition from one state to another we might end up with more than one word that is accepted on this path. The path marked with an X in figure 4.5 is such a path as it contains the transition $4 \to 8$, which accepts both 0 and 1. The resulting words on the path $1 \to 2 \to 4 \to 8 \to 10$ are therefore $\langle 0001 \rangle = 8_{10}$ and $\langle 0011 \rangle = 12_{10}$.

When we find an end state, we return the result(s) found and delete the last node we visited while backtracking. Now we are back at an earlier node and visit the next node in the neighbour list until there is no unvisited neighbour. This way we find all possible paths through the graph from the start state to one accepting end state. To get all simple paths to all end states we run this algorithm for all pairs of the start state and all end states.

We suppose that this algorithm returns a superset of constants needed to construct the semilinear set represented by the automaton. Despite much effort we could not yet find a proof that this is correct, nor could we find a counter example. We tried to find counter examples by looking at the structure of automata and tried to construct counter examples based on different structures but failed in finding one. Most of our attempts have been cases were our algorithm could find an optimized set of constants.

4.8.1. Conjecture

This section shows why this algorithm is at least a valid approach for linear sets. An automaton which accepts the solutions to a linear equation has only one final state. This follows from the construction of number decision diagrams from a linear equation as shown in [BC96]. A linear set $L(c;P)$, which can be transformed to such a linear equation, can therefore be recognized by an automaton with one

Input: Graph G, sequence of $visited$ states, end state
Output: a superset of constants

```
/* The first part of the algorithm handles the case we
   found an end state.  Reconstruct the path and return
   all possible transition sequences                    */
last ←visited.last();
neighbours ←G.nodes[last].neighbours;
for i = 0 to #neighbours do
    node ←neighbour[i];
    if node ≡ end then
        /* visited contains the full path from start to
           end                                          */
        visited.append(node);
        /* this might return more than one transition
           sequence if multiple transitions have been
           possible between two states                  */
        constants_at_path ←transitions_on_path(visited);
        constants.append(constants_at_path);
        visited.deleteLastElement();
    end
end
/* at this point we have the binary representation of
   each constant                                        */
result ←{};
for i = 0 to #constants do
    sum ←0^k;
    constant ←constants[i];
    for j = 0 to #(constant.transitions) do
        /* calculate the value of the constant from the
           transition sequence                          */
        sum ←sum + constant.transitions[j]·2^j;
    end
    result.append(sum);
end
/* now result contains integer constants               */
```

```
/* recursion, visit all neighbours                          */
for i = 0 to #neighbours do
    node ←neighbours[i];
    if visited contains node then
        /* we have already been here                        */
        continue;
    end
    visited.append(node);
    /* DepthFirst returns all results (if any).  Concat
       to the result                                        */
    result.concat(DepthFirst(G, visited, end));
    visited.deleteLastElement();
end
return result
```

Algorithm 5: Retrieving constants from automaton

final state. This linear set L has a minimal element c with respect to the lexicographic order on the vector space. This minimal element c is also the shortest accepting word in the automaton. If there would be a shorter accepting word than c, it would also be smaller than c with respect to lexicographic ordering and this element would therefore be the minimal element.

4.9. Computing a Minimal Basis

The amount of constants which have to be processed and converted to periods in the Kleene star algorithm 1 is posing a problem. If we store the constants to each automaton explicitly, then we get an exponential blowup during multiplication because the constant from each linear set has to be added to all other constants and creates a new linear set for each pair of constants (see section 3.2.3). Also if we use the algorithm 5 for retrieving constants from an automaton, we get redundant constants that are not needed and cause unnecessary work while calculating the Kleene star. For that reason we need an algorithm to minimize a superset of constants to get a minimal basis for the Kleene star algorithm as already stated in 3.2.4.2. This minimal basis is calculated with the algorithm 6 where we check for each element of the set of constants whether it is a linear combination of the other remaining constants and remove it if this is the case. This resulting set depends on the order in which the constants are checked as we remove all unnecessary elements in the for-loop. One of these removed elements might be the basis for one

of the later constants, but at least we do not have any linear dependant constants in our set and it is smaller or equal the size of the original set.

Input: list of i constants $c_i \in \mathbb{N}^k$
Output: minimal basis for the constants
for $i = 0$ **to** #*constants* **do**
    element ←constants[i];
    if *element is a linear combination of the remaining constants* **then**
        constants.remove(i);
    end
end
return *constants*;

Algorithm 6: Calculating a minimal basis from a set of constants

5. Implementation

The implementation of semilinear sets represented as NDDs is done in the C++-framework FPsolve, which has solvers for systems of equations over ω-continuous semirings. FPsolve has a very flexible architecture, which enables the user to easily switch the used solver by using template parameters, as the API for the solvers is very generic. There are also different semirings like the float semiring, the tropical semiring and even generic semirings like the tuple semiring, which can be used to combine two semirings and calculate the solutions in both semirings at the same time. Each semiring provides a generic semiring API, which allows the solver to be unaware of the actual used semiring. Still, it can use properties given by templates to distinguish for example commutative from non-commutative or idempotent from non-idempotent semirings to be able to optimize computation in certain cases. An overview of the basic architecture can be seen in figure 5.1, were we also included the structure for our semiring implementation.

5.1. GENEPI

For the actual implementation of above algorithms we used the GENEPI library, which is a *GENEric Presburger programming Interface*. The library does not do actual computing by itself, but uses a plugin system that enables the user to use different backends depending on the requirements[Gen14]. One big advantage of GENEPI is that the abstraction provides a clean and generic API which does not depend on the underlying data structure so that the user only provides a plugin for an automaton library and GENEPI is able to use it without the user needing to know about this data structure. If the user later decides that a formula based approach might get faster results, he simply exchanges the automaton based plugin with the formula based plugin and his program, which depends only on the GENEPI API, is still working.

The API offers generic functions to create data structures, which are recognizing sets like the empty set $\emptyset$ or the number sets $\mathbb{N}, \mathbb{Z}$ or $\mathbb{R}$ (depending on the underlying backend, not each set is supported). GENEPI also handles all the required data structures needed by the backend and does all the allocating and freeing of

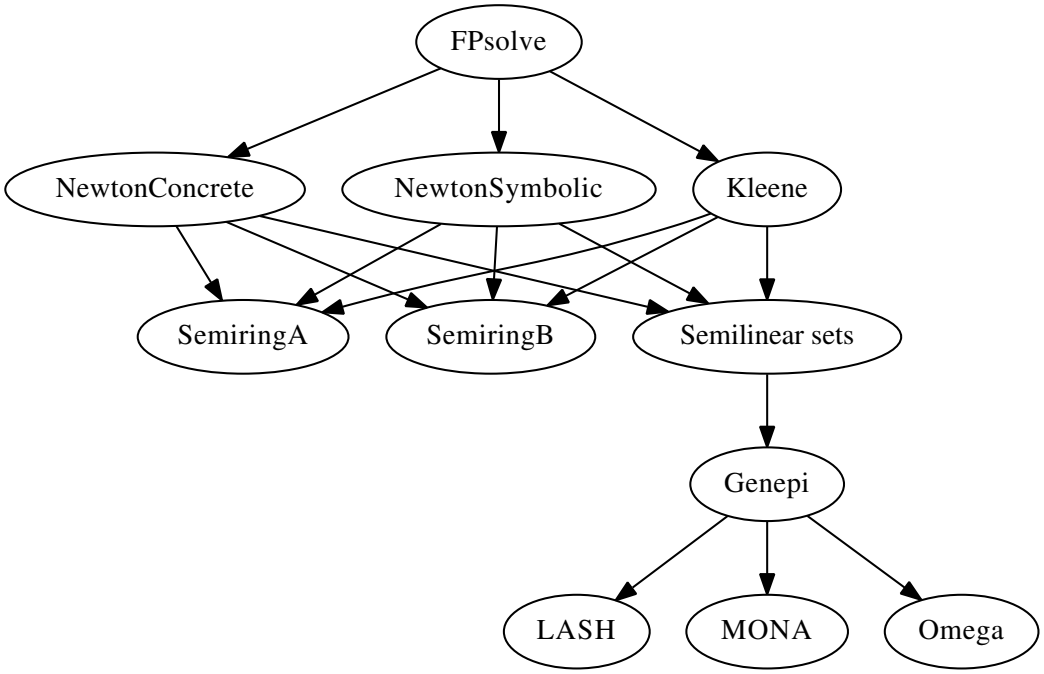


Figure 5.1.: Architecture of the semilinear set implementation

memory. In contrast to the original frameworks GENEPI checks the input for certain assertions to help the programmer avoid errors. As an example it checks if the dimension of two input automatons match each other and throws an error otherwise. In addition, GENEPI also provides some convenience functions, which are composed of small backend functions. Thus, each backend only has to provide basic functionality. As an example the equality or inclusion check in the MONA plugin is composed of `set_complement, set_intersection` and `is_empty`.

There are also methods to create automata recognizing the solutions of linear equalities or other linear operations. Once we have automata we can operate on them as well. GENEPI has functions for set intersection, union, complement, projection or inverse projection. Beside these manipulating functions it is also possible to query these automata. For example you can ask the set if it is empty, full or finite or test two sets if they are equal or included in one another. Another important function for our work is the ability to check if a vector belongs to a set.

GENEPI offers a built in plugin loader, which loads plugins during runtime in order to avoid recompiling, and functionality to check the environment for failed plugins.

5.1.1. LASH

The LASH *toolset* (Liège Automata-based Symbolic Handler) is a C-based toolset that was written around 2000 and offers data structures for several different sets. The LASH-*NDD* library (number decision diagram) can be used to recognize vectors of unbounded integers, which are slightly more expressive than Presburger Arithmetic[Boi14].

LASH-*RVA* (Real Vector Automaton) is another library in the LASH toolset, which is able to represent sets of vectors of unbounded reals or integers.

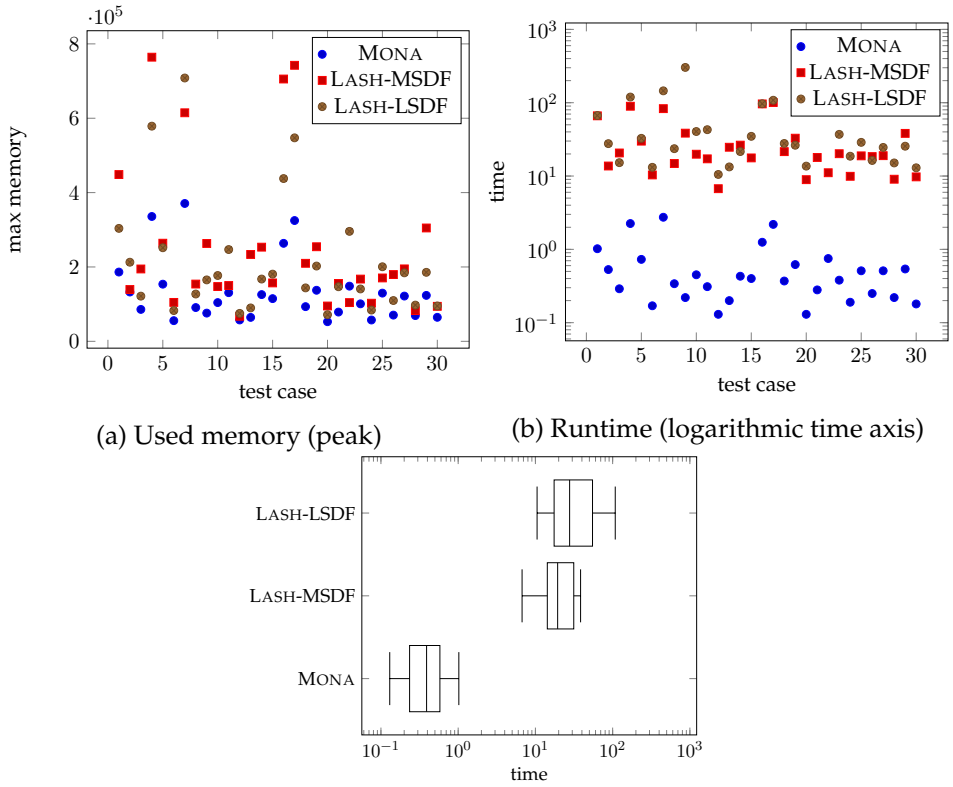
As we focus only on sets of natural numbers, we are going to use the NDD library as a GENEPI backend. The NDD library uses number decision diagrams as an internal data structure to recognize the set. It supports the LSBF and MSBF encoding of numbers and is therefore interesting for comparing operations on different encodings without the necessity of changing code.

5.1.2. MONA

MONA is a C-based tool, which can translate formulas to finite-state automata and implements decision procedures for the weak monadic second-order theory of one or two successors (WS1S/WS2S) [KM01]. To accomplish this, MONA has its own data structures for expressing and manipulating finite-state automata. These DFA implementation uses an efficient shared binary decision diagram (BDD) to store all transitions in one compact table. Although most of the code was also written around 2000, it is still maintained by the authors by fixing bugs and porting the codebase to new compilers and toolchains. MONA uses the LSBF encoding for the internal representation of the recognized vectors.

5.1.3. Comparison of LASH vs. MONA

We experimented with FPsolve and calculated the solutions to some randomly created quadratic equations with MONA and LASH (both LSBF and MSBF). The used equations for this benchmark can be found in appendix A. For this benchmark we recorded the maximum amount of memory and the time FPsolve needed to calculate the solution. Each equation was solved with our semiring implementation with each of the three GENEPI plugins MONA,LASH-LSDF and LASH-MSDF. The results that can be seen in figure 5.2 show that the MONA plugin is faster by orders of magnitude than both LASH plugins. This might come from the efficient



(a) Used memory (peak)

(b) Runtime (logarithmic time axis)

(c) Runtime (logarithmic time axis)

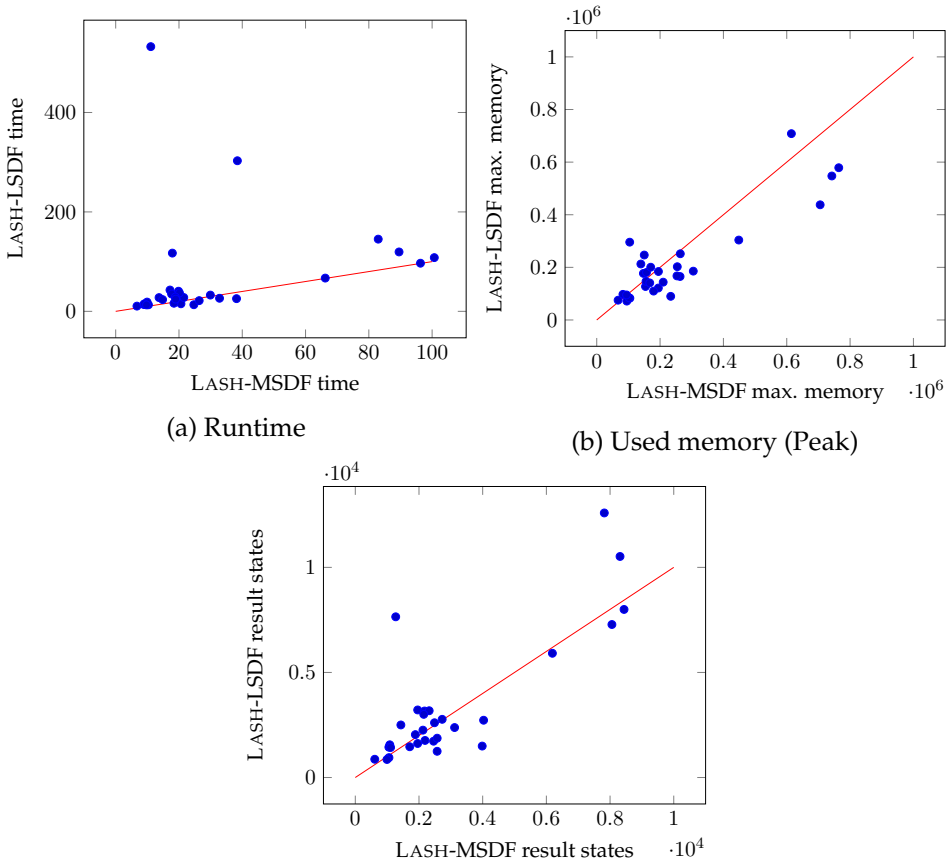
Figure 5.2.: Comparison of MONA and LASH

DFA implementation of MONA which is based on a shared BDD and other optimizations.

5.2. Optimization

5.2.1. LSBF vs. MSBF

There are examples for sets of numbers where the LSBF and MSBF representation have very different automata sizes. For the automata with basis 2, which we use in this thesis, the set of all powers of two $\{2^i | i \in [k]\}$ is representable in $\mathcal{O}(k)$ in LSBF while the MSBF automaton has a size in $\mathcal{O}(2^k)$[Boi]. Latour experimented in [Lat05] on sizes in MSBF and LSBF representation and found that the MSBF encoding produces smaller automata than the LSBF encoding. A huge impact can



(a) Runtime

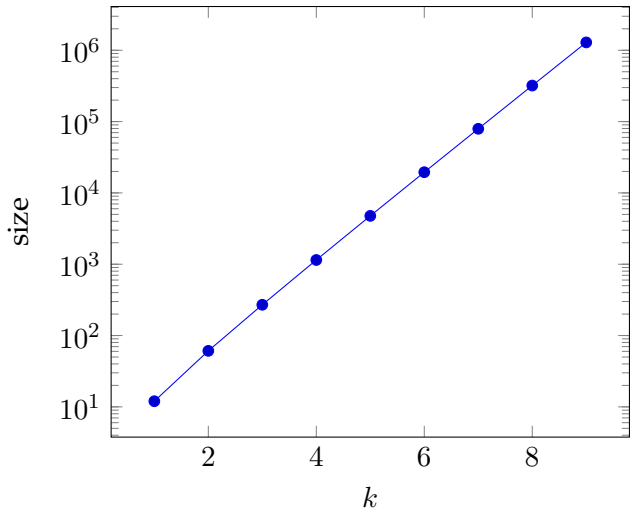
(b) Used memory (Peak)

(c) Automata states

Figure 5.3.: Comparison of LASH MSDF and LSDF plugin

be seen with the presence of a sign bit while there was almost no difference in the absence of a sign bit.

With the flexible GENEPI framework it is easily possible to change the underlying backend of our algorithms. We used the same equations as in section 5.1.3 and compared the runtime, peak memory usage and the number of states of the MSBF and LSBF encoding of the solution for each problem instance in figure 5.3 and found, that for most of our problem instances the encoding is not important. But we can see that the MSBF encoding is slightly more efficient on average both in runtime and amount of states in the solution automaton and that the LSBF can have outliers in the runtime and amount of states for a few problem instances. Despite that, the amount of used memory (peak) is about the same for both encodings.



(a) Exponential growth of automaton size

k	size
1	12
2	61
3	270
4	1147
5	4764
6	19513
7	79242
8	320119
9	1289016

(b) Table of the size of the automata

Figure 5.4.: Size of the sum automaton for k variables

5.2.2. Componentwise Addition Compared to Addition of All Components at once

As already described in section 4.6 we used an optimization for the multiplication algorithm. We have to calculate the sum of all $a_i \in A$ and $b_i \in B$ to get the resulting set. This could be done by creating one big sum automaton which calculates the sum of all pairs of a_i, b_i at once, but this automaton grows exponentially in the dimension of the recognized vectors as can be seen in figure 5.4. For this reason we do not want to generate the automaton that calculates the sum for k vector components at once. Instead we use the sum automaton for one component which is of size 12 and calculate the sum for each of the components in an extra step. Between these steps we can additionally project away the unnecessary components, which already have been used to calculate the sum, in order to keep the number of states in the intermediate automata as small as possible.

5.2.3. Order of Processing of Variables

While implementing the multiplication algorithm, some problem instances used huge amounts of memory. This happens while intersecting with the generic sum automaton as it creates automata with sizes in order of the double size of the original automaton. If we were to intersect the generic sum automaton with the next

component, the automaton would have grown to infeasible sizes. We briefly recall from section 4.6.2 that one component/variable was encoded in such a way that the encoding of the i-th variable was a_i for the first automaton a_i, b_i for the second and $a_i b_i c_i$ for the result automaton with $c_i = a_i + b_i$ after intersecting with the sum automaton. After calculating the sum c_i we do not need a_i and b_i anymore. For that reason we can project away these components and possibly reduce the amount of states and transitions in the automaton.

But we discovered that the size of the projected automata is not always smaller than the original automaton (after intersecting with the generic sum automaton). This might come from the fact that while removing information from the automaton we get a non deterministic automaton as a result which as the be converted into a deterministic automaton. The determinization can cause a blowup of the automaton and can even double the size instead of reducing it.

To circumvent this we change the order of the variables that we intersect the generic sum automaton. For some input automata the order in which we calculate the sum of the components has a big impact on the size of the intermediate automata. An unfortunate ordering might cause a huge blow up during the intersection and projection while others generate very small intermediate automata for the same input automata. But for some input automata the size of the intermediate automata are the same for each permutation of the order on the variables. Bonfante and Leroux discovered in [BLR07] that the problem of finding an optimal order for intersecting automata is NP-complete. Thus, to find a reasonably good permutation, we need a heuristic, which enables us to calculate the result without using an unreasonable amount of memory.

For finding a good heuristic we have to reflect on the information about the automaton which is available to us. We are given an automaton representing a vector of sets of numbers. To be able to calculate the Kleene star closure of the set in an efficient way, we are also given a minimal basis of the constants which gives us a rough idea about the length of the words representing the numbers for each component of the vector. We do not have more information readily available but the automaton which is kind of a black box for our purpose which we can ask for certain properties but it is an open problem which properties will provide a good heuristic for calculating the sum of the two input automata.

6. Conclusion

In this thesis we defined the Kleene star operator on semilinear sets where we only used the constants and powers of the original sets. We then transformed semilinear sets to NDDs and defined the semiring structure with addition, multiplication and the Kleene star on these automata by only using high-level operations like union, intersection, projection and inverse projection. The multiplication poses a problem because of the exponential growth in the explicit semilinear set representation. But it is also problematic in the automata representation when only using high-level operations. This is due to the intermediate automata growing very fast while the size of the result after projection would be much smaller. Experiments with the order in which vector components are summed up show that for some input instances there is a big difference in the size of the intermediate automata depending on the order. Despite our efforts we could not find a heuristic for a good order of the components. To tackle the problems of the high-level approach for the multiplications we give an algorithm for a direct construction of the resulting multiplication automaton based on the transitions of the input automata. But as our implementation uses GENEPI as an abstraction layer we cannot access the transitions of the input automata. Instead we have to rely on the high-level approach to be able to use GENEPI, which has the big advantage that we can switch the backend without effort. While the direct construction approach does not create as many intermediate automata we still get a huge and non-deterministic automaton, which has to be determinized and minimized. Finally we describe an algorithm which we suppose extracts a superset of necessary constants of the semilinear set from the automata representing these sets but could not prove yet.

7. Future work

7.1. Heuristic for a Good Processing Order during Multiplication

The high-level multiplication algorithm still poses a problem regarding the order in which we calculate the sum of the components of our input automata (see section 5.2.3). At the moment we add up the components in their order of appearance in the input vector but as we mentioned there might be a better order in which we do the necessary automata intersections. We were not able to find a good heuristic to calculate such an order. Perhaps there is enough data in the automata which is efficiently extractable to calculate a good order. Another possible way would be to store some meta data with the automaton which is updated during addition, multiplication and the Kleene star which gives us enough information.

7.2. Extracting Constants from Automatons

In section 4.8 we describe an algorithm which might extract necessary constants from a given automaton that are needed to recreate the represented semilinear set. It is still an open question to define which constants are necessary when we do not know the periods which are still hidden in the automaton. After defining which constants are necessary it might be possible to prove the algorithm. Additionally, it would be also of interest to extract only a minimal set of constants, which define the same semilinear set with the hidden periods in the automaton.

7.3. Direct Creation of Multiplication Automaton

As already mentioned the high-level approach for creating the multiplication automaton is not very efficient because many very big intermediate automata are created during the calculation of the result. One possible solution is the direct

construction of the resulting multiplication automaton by not using black-box automata but directly looking at the internal data structure and create the result directly without creating several huge intermediate data structures. But this has the disadvantage that we cannot use the flexibility of GENEPI anymore as GENEPI does not support working directly on the automaton with its API. For this approach one has to choose an automata framework and use it without the GENEPI abstraction layer. Another approach is to extend GENEPI with a middle-level API which gives the programmer generic access to the structure of the automata to get transitions between states without touching the internal data structures. These data structures can be complex and optimized. For example the data structures used in MONA encode the transitions of an automaton in a very efficient way. They are based on binary decision diagrams[KM01], which is quite different to the LASH DFA implementation. The solution with a generic middle-level API that enables the programmer to extract transitions and create new ones would be preferable. This is because the internal optimizations (i.e. the BDD structure of MONA) can be used without having to operate directly on its internal data structure. In addition, it is still possible to switch the backend to experiment with different automata libraries.

Appendix

A. FPsolve Benchmark for Semilinear Sets

These 30 test cases can be used as input for the tool FPsolve. They have been randomly generated, such that we have quadratic equations with at most 4 letters. We calculated the least fixed point of these equations with the Newton Concrete solver of FPsolve and measured the memory consumption, runtime and the size of the solution for each of the three plugins MONA,LASH-MSDF and LASH-LSDF.

```
1 : <x0> ::= "<c:1,c:6,d:1,b:5>"<x0><x0> | \
         "<a:5,a:4,a:6,d:4>"<x0><x0> | "<d:3,c:2,b:2,b:3>"<x0><x0> | "<>";
2 : <x0> ::= "<a:4,b:6,d:6,b:3>"<x0><x0> | \
         "<c:4,a:1,d:1,b:6>"<x0><x0> | "<a:1,d:5,b:3,a:1>"<x0><x0> | "<>";
3 : <x0> ::= "<d:1,c:3,c:5,c:2>"<x0><x0> | \
         "<b:2,d:6,a:2,d:6>"<x0><x0> | "<a:4,c:3,c:1,a:6>"<x0><x0> | "<>";
4 : <x0> ::= "<b:4,b:3,b:2,a:6>"<x0><x0> | \
         "<a:5,c:5,c:2,c:5>"<x0><x0> | "<a:4,c:3,d:5,a:4>"<x0><x0> | "<>";
5 : <x0> ::= "<d:5,a:1,a:6,a:2>"<x0><x0> | \
         "<c:6,a:3,a:2,c:5>"<x0><x0> | "<a:6,a:5,c:1,a:2>"<x0><x0> | "<>";
6 : <x0> ::= "<a:5,d:2,a:2,a:5>"<x0><x0> | \
         "<c:5,d:5,a:3,a:6>"<x0><x0> | "<a:6,c:2,d:3,c:4>"<x0><x0> | "<>";
7 : <x0> ::= "<b:6,a:2,b:6,b:5>"<x0><x0> | \
         "<d:5,b:3,a:3,a:2>"<x0><x0> | "<d:2,c:5,c:5,d:1>"<x0><x0> | "<>";
8 : <x0> ::= "<a:3,c:2,c:2,c:6>"<x0><x0> | \
         "<a:4,a:1,c:2,d:4>"<x0><x0> | "<d:5,a:4,d:5,d:3>"<x0><x0> | "<>";
9 : <x0> ::= "<b:6,d:1,d:3,c:3>"<x0><x0> | \
         "<c:4,c:1,d:2,d:6>"<x0><x0> | "<d:6,b:5,c:3,c:1>"<x0><x0> | "<>";
10: <x0> ::= "<b:4,a:1,b:5,a:1>"<x0><x0> | \
         "<a:1,a:4,d:6,c:6>"<x0><x0> | "<b:4,b:5,b:4,c:4>"<x0><x0> | "<>";
11: <x0> ::= "<b:6,b:1,a:3,a:5>"<x0><x0> | \
         "<c:6,c:1,c:6,b:5>"<x0><x0> | "<c:5,c:1,c:6,b:3>"<x0><x0> | "<>";
12: <x0> ::= "<d:5,a:5,a:1,a:2>"<x0><x0> | \
         "<a:4,d:2,c:4,d:6>"<x0><x0> | "<b:2,c:4,a:3,d:2>"<x0><x0> | "<>";
13: <x0> ::= "<c:2,d:3,b:4,a:3>"<x0><x0> | \
         "<c:3,a:3,c:4,c:1>"<x0><x0> | "<d:6,d:6,b:4,d:4>"<x0><x0> | "<>";
14: <x0> ::= "<d:3,d:1,d:4,a:3>"<x0><x0> | \
         "<c:5,a:2,a:6,b:5>"<x0><x0> | "<a:5,a:6,d:4,d:2>"<x0><x0> | "<>";
15: <x0> ::= "<a:3,c:2,b:6,d:1>"<x0><x0> | \
         "<b:4,d:6,b:2,a:3>"<x0><x0> | "<c:5,c:2,b:1,b:3>"<x0><x0> | "<>";
```

```
16: <x0> ::= "<c:5,d:4,b:6,b:3>"<x0><x0> | \
         "<d:6,d:2,d:2,a:5>"<x0><x0> | "<b:2,a:5,c:5,b:4>"<x0><x0> | "<>";
17: <x0> ::= "<d:6,b:5,b:5,a:3>"<x0><x0> | \
         "<a:6,a:5,b:3,b:4>"<x0><x0> | "<d:5,a:4,d:6,d:2>"<x0><x0> | "<>";
18: <x0> ::= "<a:1,c:4,d:1,d:5>"<x0><x0> | \
         "<a:1,b:2,d:6,b:5>"<x0><x0> | "<b:6,a:6,c:4,a:4>"<x0><x0> | "<>";
19: <x0> ::= "<c:2,a:1,b:4,c:4>"<x0><x0> | \
         "<d:3,d:4,c:1,c:5>"<x0><x0> | "<d:6,d:5,b:1,d:1>"<x0><x0> | "<>";
20: <x0> ::= "<c:5,c:4,a:6,b:2>"<x0><x0> | \
         "<c:2,a:4,c:1,c:1>"<x0><x0> | "<d:1,b:6,a:5,c:3>"<x0><x0> | "<>";
21: <x0> ::= "<c:1,d:1,d:4,a:5>"<x0><x0> | \
         "<d:2,d:4,d:4,c:5>"<x0><x0> | "<d:6,c:5,d:6,a:2>"<x0><x0> | "<>";
22: <x0> ::= "<c:3,d:3,c:4,b:6>"<x0><x0> | \
         "<b:4,d:1,d:1,d:4>"<x0><x0> | "<b:2,b:2,c:6,c:5>"<x0><x0> | "<>";
23: <x0> ::= "<a:5,c:6,c:6,d:3>"<x0><x0> | \
         "<c:4,d:1,d:2,a:5>"<x0><x0> | "<c:5,b:5,a:6,c:2>"<x0><x0> | "<>";
24: <x0> ::= "<c:6,c:4,a:1,c:2>"<x0><x0> | \
         "<b:6,c:6,c:2,c:2>"<x0><x0> | "<a:2,c:5,b:3,a:1>"<x0><x0> | "<>";
25: <x0> ::= "<a:4,d:1,b:2,d:2>"<x0><x0> | \
         "<b:3,a:1,b:6,c:6>"<x0><x0> | "<a:1,b:2,a:6,c:2>"<x0><x0> | "<>";
26: <x0> ::= "<c:4,a:2,b:6,c:2>"<x0><x0> | \
         "<a:5,a:1,b:5,d:2>"<x0><x0> | "<c:6,a:4,a:1,c:5>"<x0><x0> | "<>";
27: <x0> ::= "<c:2,c:4,b:2,d:4>"<x0><x0> | \
         "<d:5,b:3,b:2,b:4>"<x0><x0> | "<d:6,c:1,b:1,d:5>"<x0><x0> | "<>";
28: <x0> ::= "<a:1,c:1,d:3,c:1>"<x0><x0> | \
         "<b:6,c:4,a:5,a:4>"<x0><x0> | "<d:4,b:5,a:5,b:1>"<x0><x0> | "<>";
29: <x0> ::= "<b:6,a:1,d:3,c:2>"<x0><x0> | \
         "<c:5,a:5,d:5,b:6>"<x0><x0> | "<d:1,c:1,c:4,c:4>"<x0><x0> | "<>";
30: <x0> ::= "<b:3,c:2,a:1,a:4>"<x0><x0> | \
         "<d:4,b:5,b:2,a:1>"<x0><x0> | "<c:3,c:6,b:2,c:1>"<x0><x0> | "<>";
```

List of Figures

List of Algorithms

Bibliography

[BC96] Alexandre Boudet and Hubert Comon. Diophantine equations, presburger arithmetic and finite automata. In Hélène Kirchner, editor, *Trees in Algebra and Programming CAAP '96*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer Berlin Heidelberg, 1996.

[BLR07] Guillaume Bonfante and Joseph Le Roux. Intersection Optimization is NP-Complete. In *Sixth International Workshop on Finite-State Methods and Natural Language Processing - FSMNLP 2007*, Postdam, Germany, 2007.

[Boi] Bernard Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis.

[Boi14] Bernard Boigelot. The lash toolset. `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`, May 2014.

[DKV09] Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of Weighted Automata*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[EGKL11] Javier Esparza, Pierre Ganty, Stefan Kiefer, and Michael Luttenberger. Parikhs theorem: A simple and direct automaton construction. *Information Processing Letters*, 111(12):614 – 619, 2011.

[EKL07] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. On fixed point equations over commutative semirings. In Wolfgang Thomas and Pascal Weil, editors, *STACS 2007*, volume 4393 of *Lecture Notes in Computer Science*, pages 296–307. Springer Berlin Heidelberg, 2007.

[EKL10] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. Newtonian program analysis. *Journal of the ACM*, 57(6):33:1–33:47, October 2010.

[EP02] Katrin Erk and Lutz Priese. *Theoretische Informatik*. Springer-Lehrbuch. Springer, 2002.

[FPs14] FPsolve. Solver for polynomial equations over ω-continuous semirings. `https://github.com/mschlund/FPsolve`, May 2014.

[FR79] Jeanne Ferrante and Charles W Rackoff. *The computational complexity of logical theories*. Lecture Notes in Mathematics. Springer, Berlin, 1979.

[Gen14] Genepi. Generic presburger programming interface. `http://tapas.labri.fr/trac/wiki/GENEPI`, May 2014.

[Gin66] Seymour Ginsburg. *The Mathematical Theory of Context-Free Languages*. McGraw-Hill, Inc., New York, NY, USA, 1966.

[GS66] Seymour Ginsburg and Edwin H. Spanier. Semigroups, presburger formulas, and languages. *Pacific Journal of Mathematics*, 16(2):285–296, 1966.

[HK71] J. Hopcroft and R. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 0, Dept. of Computer Science, Cornell U, December 1971.

[Kle52] S. Kleene. Introduction to Metamathematics, 1952.

[KM01] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, Aarhus University, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3.

[Koz94] Dexter Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and Computation*, 110:366–390, 1994.

[Kui97] Werner Kuich. Semirings and formal power series: Their relevance to formal languages and automata. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, pages 609–677. Springer Berlin Heidelberg, 1997.

[Lat04] L. Latour. From automata to formulas: convex integer polyhedra. In *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 120–129, July 2004.

[Lat05] Louis Latour. *Presburger Arithmetic: From Automata to Formulas*. PhD thesis, Université de Liège, 2005.

[Ler05] Jerome Leroux. A polynomial time presburger criterion and synthesis for number decision diagrams. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, LICS '05, pages 147–156, Washington, DC, USA, 2005. IEEE Computer Society.

[LS13] Michael Luttenberger and Maximilian Schlund. Convergence of newtons method over commutative semirings. In Adrian-Horia Dediu, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications*, volume 7810 of *Lecture Notes in Computer Science*, pages 407–418. Springer Berlin Heidelberg, 2013.

[Lug04] Denis Lugiez. From Automata to Semilinear Sets: a Solution for Polyhedra and Even More General Sets. Research report 21-2004, LIF, Marseille, France, april 2004. `http://pageperso.lif.univ-mrs.fr/~edouard.thiel/RESP/Rapports/21-2004.html`.

[Lug05] Denis Lugiez. From automata to semilinear sets: A logical solution for sets $\mathcal{L}(\mathcal{C}, \mathcal{P})$. In Michael Domaratzki, Alexander Okhotin, Kai Salomaa, and Sheng Yu, editors, *Implementation and Application of Automata*, volume 3317 of *Lecture Notes in Computer Science*, pages 321–322. Springer Berlin Heidelberg, 2005.

[Muc03] An. A. Muchnik. The definable criterion for definability in presburger arithmetic and its applications. *Theor. Comput. Sci.*, 290(3):1433–1444, January 2003.

[Par66] Rohit J. Parikh. On context-free languages. *J. ACM*, 13(4):570–581, October 1966.

[Sta84] Ryan Stansifer. Presburgers article on integer airthmetic: Remarks and translation. Technical Report TR84-639, Cornell University, Computer Science Department, September 1984.

[STL13] Maximilian Schlund, Michał Terepeta, and Michael Luttenberger. Putting newton into practice: A solver for polynomial equations over semirings. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 8312 of *Lecture Notes in Computer Science*, pages 727–734. Springer Berlin Heidelberg, 2013.