COUNTER AUTOMATA AND CLASSICAL LOGICS FOR DATA WORDS

By

Amaldev Manuel

The Institute of Mathematical Sciences, Chennai.

A thesis submitted to the
Board of Studies in Mathematical Sciences

in partial fulfillment of the requirements
for the Degree of

DOCTOR OF PHILOSOPHY

of

HOMI BHABHA NATIONAL INSTITUTE

October 2011

Homi Bhabha National Institute

Recommendations of the Viva Voce Board

As members of the Viva Voce Board, we recommend that the dissertation prepared by **Amaldev Manuel** entitled "Counter Automata and Classical Logics for Data Words" may be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

-- **Date :**
Chairman : R. Ramanujam

-- **Date :**
Convener : V. Arvind

-- **Date :**
Member : Kamal Lodaya

-- **Date :**
Member : Madhavan Mukund

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to HBNI.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it may be accepted as fulfilling the dissertation requirement.

-- **Date :**
Guide : R. Ramanujam

DECLARATION

I hereby declare that the investigation presented in the thesis has been carried out by me. The work is original and the work has not been submitted earlier as a whole or in part for a degree/diploma at this or any other Institution or University.

Amaldev Manuel

ACKNOWLEDGEMENTS

First and foremost I thank my guide R. Ramanujam who supported me throughout my doctoral years with his vast knowledge, sublime perspectives and unearthly patience while giving me freedom to work in my own way. He has been a great guide, an exemplary teacher and a marvelous friend to me both in the darkest hours and brightest days.

I am grateful to all the faculty members in Matscience and CMI especially to Madhavan Mukund, Kamal Lodaya, K. Narayan Kumar and S. P. Suresh. Thanks to Benedikt Loewe for hosting me in Amsterdam and Maarten Marx for mentoring me. Thanks Paritosh Pandya for allowing me to spend many summers in TIFR. I am grateful to Luc Segoufin and Thomas Schwentick for their great suggestions and feed-backs. Thanks to my co-author Thomas Zeume.

I thank my all friends in Matscience especially Bruno (now Brenda), Muthu, Sheeraz, Phawade, Suman, Sylvester and George for their great company and support. Also, thanks to Ajesh and Benny for making my summers in TIFR wonderful. Thanks to Biju for being my constant companion during my stay in Netherlands. Special thanks to David for the great discussions I had with him over these years.

I thank my parents, parents-in-law, sister, brother and brothers-in-law for their encouragement and support. Finally, thanks to my wife for proofreading my papers, giving valuable suggestions and criticisms and above all for her unmatched love and patience.

Abstract

This thesis takes shape in the ongoing study of automata and logics for data words – finite words labelled with elements from an infinite alphabet. The notion of data words is a natural way for modelling unboundedness arising in different areas of computation. The contribution of this thesis is two-fold, which we discuss briefly below.

On the automata side, after introducing two known models – Register automata and Data automata – we formulate a model of computation for data words, namely Class Counting Automaton (CCA). CCA is a finite state automaton equipped with countably infinitely many counters where counters can be increased or reset. Decrement is not allowed to preserve decidability. We prove basic facts about this model and compare its expressive power with respect to the earlier models. It is shown that this automaton sits (roughly) in between register automata in terms of expressiveness and complexity of decision problems. We also study several extensions some of which subsume earlier models.

In the second part we look at the two-variable logics (first-order logic restricted to two variable) on logical structures which correspond to data words, continuing the study initiated in [BDM$^+$11]. First, it is shown that two-variable logic on structures with two linear orders and their successor relations is undecidable. Then we consider first-order structures with successors of two linear orders and show that finite satisfiability of two-variable logic is decidable on these structures. We use suitably defined automata for proving this result. Later, we generalize the above proof to the case of k-bounded ordered data words – first-order structures with a linear successor and a total preorder with an additional restriction of k-boundedness on the preorder – and prove a similar result. A corollary of this result is that two-variable logic is decidable on structures with two successors and at most one order relation. The decidability results are sharpened by showing lower bounds for decidable fragments and exhibiting undecidability results for richer fragments.

Contents

List of Figures

1

Introduction

The formalism of languages over finite alphabets is well-suited for abstracting sequential behaviour of computing systems such as execution traces of programs and plays of games. Hence the multipronged — algebraic, logical, automata-theoretic — study of languages over finite alphabets has contributed effectively to the verification of software and control systems. A standard approach is the following. The program or control system under scrutiny is abstracted as a finite state system and the specification is written in a specification language, very often a suitable logic. It is then checked that the finite state system obeys the specification by comparing the languages described by the system and the specification. A necessary premise for this method is the effectiveness of checking non-emptiness of the language defined by the system and checking satisfiability of the formula. The above approach to program verification, called model checking [CGP99], has been found very effective in the verification of stand-alone reactive systems like control systems in automobiles.

This thesis takes shape in an ongoing effort to find suitable formalisms, both automata theoretic and symbolic, for languages over *infinite alphabets*. Infinite alphabets are an obvious way to abstract unboundedness often occurring in many areas of computer science. Natural examples are values of variables in programs, process IDs in distributed computing, nonces in security protocols, attribute values in XML, keys in data bases etc. Apparent uses of such mechanisms are many fold [MRR+08], the single most important application being in verification.

1.1 Words over infinite alphabets

Let Σ be a finite alphabet and Γ be an infinite set in which membership and equality are decidable. We call finite sequences of elements of the set $\Sigma \times \Gamma$ *data words*. Formally a data word $w = (a_1, d_1) \dots (a_n, d_n)$ is in $(\Sigma \times \Gamma)^*$. A data language is a set of such words.

The course of study of data languages so far has been driven by two important questions, which are: (1) what is a suitable class of finite state automata for recognizing data languages with a decidable emptiness problem? (2) what is a suitable logical language for expressing data languages with a decidable satisfiability problem? The contributions of this thesis are to be seen in the light of these two questions which we discuss briefly below.

1.2 Automata for data words

We mentioned above that the effectiveness of finite state model checking is expediated by the presence of a class of languages captured by the notion of regularity. In the case of finite words regularity is synonymous with the confluence of the following properties: closure under boolean and other natural operations, low complexity of the decision problems such as membership and non-emptiness, alternate characterizations in terms of logics and algebras and robustness in terms of machine characterization in the sense that the restriction of determinism or the addition of alternation or two-way-ness does not break the characterization.

An important question is whether there is a regular class of data languages. As of now the literature does not possess such a class. Of the above properties the decidability of non-emptiness problem plays a pivotal role in verification. Hence, unsurprisingly a good amount of time and energy has been invested in finding classes of automata with decidable non-emptiness problem.

The general approach, so far, for designing automata for data words has been to augment a finite state automaton with memory structures. This idea traces its origin to the dawn of automata theory in the fifties and sixties when an intensive study of automata with various memory structures such as stacks, queues, push-downs, counters, tapes etc. was performed. Following this line, most important classes of automata known for data words employ structures such as registers, hash tables, counters, stacks, pointers etc.

Among the various automaton models proposed for data languages, two, Register automata and Data automata got particular attention. A Register automaton [KF94, DL09] is a finite state automaton equipped with a finite number of registers which can hold data values. The transitions depend on the state of the automaton as well as the register configuration. It is easy to observe that since the registers are only finitely many the automaton is unable to keep track of all the data values it has seen, thus incapable of recognizing the language "all data values occurring in the word are different". However register automata are akin to finite state automata in the sense that the string projections of the language accepted by a register automaton is regular. The nonemptiness problem of register automata is NP-complete. A Data automaton [BDM$^+$11], is a composition of two finite state machines where regular properties over the entire word and over data values can be checked. They strictly subsume register automata in terms of the set of accepted languages. They are capable of accepting languages like "all data values under a are distinct", "every data value occurring under a occurs under b" etc. However their nonemptiness problem is of very high complexity (not known to be elementary). Neither of the above classes of automata is complementable.

Our approach to the automaton problem involves enhancing finite state automata with counters. Counters are a primitive and minimal computational device where the operations are increment, decrement and checking for zero. Yet it is long been known that automata with two counters are as powerful as Turing machines. Hence it is necessary to restrict the operations on the counters. There are standard restrictions in the literature. Some of them are: (1) disallowing the decrement operation, (2) removing the two-way branching on a zero test, (3) allowing counter values to be negative.

In this thesis we introduce a class of machines we call Class Counting Automata. A class counting automaton $A = (Q, \Sigma, \Delta, I, F)$ is a finite state automaton with $|\Gamma|$-many counters, where Q is the finite set of states, Δ is the transition relation and $I \subseteq Q$ and $F \subseteq Q$ are the set of initial and final set of states. A configuration

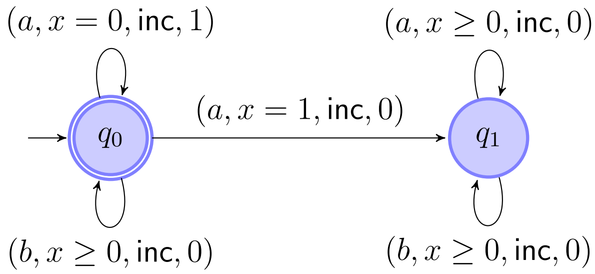

Figure 1.1: CCA accepting the language "All data values under a are distinct"

of the automaton is of the form (q, h) where $q \in Q$ and $h : \Gamma \to \mathbb{N}$ is a function holding the counter values. The transitions of the automaton are of the form $(p, a, \varphi(x), u, q)$ where p, q are the entry and exit states of the transition, $\varphi(x)$ is a univariate linear inequality and u is from the set $\{\mathsf{inc}, \mathsf{reset}\}$. The intended semantics of the transition is that on a configuration (p, h) of the automaton on the pair (a, d) the transition $(p, a, \varphi(x), u, q)$ can be taken if $\varphi(h(d))$ is true. The resulting configuration will be (q, h') where h' is h for all but d where $h'(d) = h(d) + 1$ if u is inc and $h'(d) = 0$ if reset.

Theorem 1.2.1. *The nonemptiness problem of CCA is* Expspace*-complete.*

Note that the complexity is to be contrasted with that of register automata (NP-complete) and that of data automata (not known to be elementary). The model checking problem for CCA is NP-complete. Addition of alternation or twowayness leads to undecidability of the nonemptiness problem.

CCA are closed under union and intersection but they are not closed under complementation. The deterministic fragment is closed under complementation but is strictly less powerful. It is not known whether they subsume register automata.

We also study several extensions and restrictions of class counting automata which are equivalent to Register automata and Data automata.

1.3 Logics for data words

Various modal and classical logics are used for specifying properties over words over a finite alphabet. On the classical side, monadic second order logic and first order logic are the most important ones, while modal languages, very attractive due to their lower complexity and intuitiveness, include linear and branching time temporal logics.

For the purpose of verification the most important aspect regarding a logic is the decidability of the model checking and the satisfiability. A whole lot of other questions reduce to checking satisfiability, for example checking implication between two formulas, checking validity of a formula etc. From a practical point of view the finite satisfiability problem ("is there a *finite* model satisfying the formula?") is more interesting than the general problem.

This thesis focuses on classical logics on data words. For this purpose, data words can be represented as a first order structure $w = ([n], \Sigma, <, +1, \sim)$ extending the corresponding representation for words due to Büchi. Here $[n]$ denotes the set $\{1, \ldots, n\}$, and Σ stands for the unary relations indicating the alphabet labelling. The binary relations $<, +1$ are interpreted as the natural linear order and successor relations on the set $[n]$. The binary relation $\sim$ denotes the equivalence relation on $[n]$ given by the data values based on equality. That is to say, $i \sim j$ if $d_i = d_j$. In addition if we have a linear order $<_\Gamma$ on the data values then this uniquely defines a total preorder $<_p$ (a total preorder is a reflexive, transitive and total binary relation) on the positions $[n]$. We denote the successor relation of $<_p$ by $+1_p$. In the following we denote linear orders and their successor relations by $<_{l_1}, <_{l_2}, \ldots$ and by $+1_{l_1}, +1_{l_2}, \ldots$.

It is easy to see that satisfiability and finite satisfiability of first order logic on data words, $\mathrm{FO}(\Sigma, <, \sim)$ is undecidable. The problems remain undecidable even for the fragment $\mathrm{FO}^3(\Sigma, <, \sim)$, the set of formulas which uses at most 3 variables. Hence for decidability one has to look for suitable restrictions which are sufficiently expressive. The two-variable fragment is a natural candidate. It is known that satisfiability problem for first order logic with two variables is decidable [Mor75, GKV97]. Since it is not expressible in FO^2 that a binary relation R is a linear order (or an equivalence relation or a preorder), the above theorem does not imply satisfiability of FO^2 over data words or over ordered data words. In a landmark paper [BDM$^+$11] it was shown that;

Theorem 1.3.1 ([BDM$^+$11]). *Finite satisfiability of* $\mathrm{FO}^2(\Sigma, <_{l_1}, +1_{l_1}, \sim)$ *is decidable.*

Note that $+1_{l_1}$ is not definable in terms of $<_{l_1}$ using two-variables over words. This prompts us to add both the order and successor relations of the linear order to the vocabulary. [As a side note, it is also the case that $+1_{l_1}$ is not definable in terms of $<_{l_1}$ and $\sim$ using two-variables over words.] Decidability holds even when $<_{l_1}$ is the ordinal ω. Status of the infinite satisfiability problem is not known.

However, the theorem fails for ordered data words;

Proposition 1.3.2 ([BDM+11])**.** *Finite satisfiability problems of* $\mathrm{FO}^2(\Sigma, <_{l_1}, +1_{l_1}, <_{p2}$ $)$ *and* $\mathrm{FO}^2(\Sigma, <_{l_1}, +1_{l_1}, +1_{p2})$ *are undecidable. In fact, undecidability persists even when the equivalence classes of* $<_{p_2}$ *are of size at most* 2*.*

This implies that in the presence of a total order on data values to get back decidability either $<_{l_1}$ or $+1_{l_1}$ has to be dropped from the vocabulary. The former case was undertaken in [SZ10] where it was shown that $\mathrm{FO}^2(\Sigma, <_{l_1}, <_{p_2}, +1_{p_2})$ is decidable. We consider the latter case when the preorder is in fact a linear order (in the case of data words it corresponds to the scenario when all the data values are distinct) and show that,

Theorem 1.3.3. *Finite satisfiability of* $\mathrm{FO}^2(\Sigma, +1_{l_1}, +1_{l_2})$ *is decidable.*

Proposition 1.3.4. *Finite satisfiability of* $\mathrm{FO}^2(\Sigma, <_{l_1}, +1_{l_1}, <_{l_2}, +1_{l_2})$ *is undecidable.*

Note that this line of work is interesting on its own [Ott01]. Our proof is automata theoretic. Concurrently, it was shown that removing at least one successor relation also results in decidability[SZ10], that is;

Theorem 1.3.5 ([SZ10])**.** *Finite satisfiability of* $\mathrm{FO}^2(\Sigma, <_{l_1}, +1_{l_1}, <_{l_2})$ *is decidable.*

This raises the question whether FO^2 is decidable if one of the order relations is absent from the vocabulary. We answer this question positively. In fact, a more general theorem is proved which says that $\mathrm{FO}^2\,(\Sigma, +1_{l_1}, <_{p_2}, +1_{p_2})$ is decidable where $+1_{l_1}$ is a successor of a linear order and $<_{p_2}, +1_{p_2}$ are a total preorder and its successor relation where the equivalence classes of the preorder is bounded by a constant. This is to be contrasted with Proposition 1.3.2.

Theorem 1.3.6. *Fix $k \in \mathbb{N}$. Finite satisfiability of* $\mathrm{FO}^2(\Sigma, +1_{l_1}, <_{p_2}, +1_{p_2})$ *is decidable when classes of $<_{p_2}$ are of size at most k.*

For the proof, the notion of data automata are generalized so that they accept ordered data words. A translation from the above logic to these automata is established and finally the non-emptiness of these automata are shown to be decidable by reduction to reachability problem in vector addition systems. Since it is definable in FO^2 that $<_{p_2}$ is a linear order, this implies the answer to the previous question.

Corollary 1.3.7. *Finite satisfiability of* $\mathrm{FO}^2(\Sigma, +1_{l_1}, <_{l_2}, +1_{l_2})$ *is decidable.*

Though the problem is decidable, it turns out to be as hard as reachability in vector addition systems.

1.4 Organization of the thesis

In Chapter 2 we recapitulate the necessary definitions and theorems required for the rest of the thesis.

In Chapter 3 Register automata and Data automata are introduced and major facts about them are briefly surveyed.

In Chapter 4 we introduce the model of Class Counting automata and give examples. Some extensions of Class counting automata are also detailed. The decidability issues of class counting automata and its variants are studied.

In Chapter 5 we introduce first order logic over data words and show basic undecidability results. The landmark results on two-variable logic over data words are outlined.

In Chapter 6 it is shown that two-variable logic with two successor relations is decidable.

In Chapter 7 two-variable logic on k-bounded ordered data words is studied. A number of undecidability results are also proved here.

In Chapter 8 we summarize by a comparison of automaton models in terms of expressiveness and complexity of nonemptiness problems. The complexity of satisfiability problems of the logics is discussed.

2

Preliminaries

In this chapter we recapitulate some definitions and theorems used in the later chapters. Let $k > 0$; we use $[k]$ to denote the set $\{1, 2, \ldots k\}$. When we say $[k]_0$, we mean the set $\{0\} \cup [k]$. By $\mathbb{N}$ we mean the set of natural numbers $\{0, 1, \ldots\}$. When $f : A \to B$, $(a, b) \in (A \times B)$, by $f \oplus (a, b)$, we mean the function $f' : A \to B$, where $f'(a') = f(a')$ for all $a' \in A, a' \neq a$, and $f'(a) = b$.

2.1 Automata Formalisms

We recall the definitions of some existing automaton models.

2.1.1 Finite state automata

Definition 2.1.1. *A finite state automaton A is a tuple $A = (Q, \Sigma, \Delta, I, F)$ where Q is a finite set of states, Σ is a finite alphabet, $\Delta \subseteq (Q \times \Sigma \times Q)$ is the set of transitions, $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final states.*

Given a word $w = a_1 \ldots a_n \in \Sigma^*$, a run ρ of A over w is a sequence $q_0 \ldots q_n$ such that $q_0 \in I$ and for all $1 \leq i \leq n$, $(q_{i-1}, a_i, q_i) \in \Delta$. The run ρ is accepting if $q_n \in F$. The language of A, denoted by $L(A)$, is the set of words w such that A has an accepting run on w.

Languages accepted by finite state automata are closed under union, intersection, complementation, homomorphisms and inverse homomorphisms.

2.1.2 Finite state transducers

Definition 2.1.2. *A finite state letter-to-letter transducer A is given by the tuple $A = (Q, \Sigma, \Sigma', \Delta, O, I, F)$, where Q is a finite set of states, Σ is a finite input alphabet, Σ' is a finite output alphabet, $\Delta \subseteq (Q \times \Sigma \times Q)$ is the set of transitions, $O : \Delta \to \Sigma'$ is the output function, $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final states.*

For $\delta = (p, a, q) \in \Delta$, we denote the output state of δ by $\mathsf{output}\,(\delta) = q$ and the input state of δ by $\mathsf{input}\,(\delta) = p$.

Given a word $w = a_1 \dots a_n \in \Sigma^*$, a run ρ of A over w is a sequence of transitions $\delta_1 \dots \delta_n$ such that δ_1 is a transition from a state in I (that is to say, $\mathsf{input}\,(\delta_1) \in I$) and for every $1 \leq i \leq n$ the output state of δ_{i-1} and the input state of δ_i are the same (that is to say, $\mathsf{output}\,(\delta_{i-1}) = \mathsf{input}\,(\delta_i)$). The run ρ is accepting if $\mathsf{output}\,(\delta_n) \in F$. A successful run ρ defines a unique output word $w' = O(\delta_1) \dots O(\delta_n)$ over the alphabet Σ'.

2.1.3 Petri nets

Definition 2.1.3. *A Petri net N is a tuple $N = (S, T, F, M_0)$, where S is a finite set of places, T is a finite set of transitions such that S and T are disjoint, $F \subseteq (S \times T) \cup (T \times S)$ is a set of flows, and $M_0 : S \to \mathbb{N}$ is the initial marking.*

The preset of a transition $t \in T$ is the set of its input places: ${}^\bullet t = \{s \in S \mid (s, t) \in F\}$; its postset is the set of its output places: $t^\bullet = \{s \in S \mid (t, s) \in F\}$. Definitions of pre- and postsets of places are analogous.

A marking of a Petri net is a multiset of its places, that is a mapping $M : S \to \mathbb{N}$. We say the marking assigns to each place a number of tokens. Markings can be added in the following way. Let M_1 and M_2 be markings.

$$M_1 + M_2 = \{(s, (M_1(s) + M_2(s))) \mid s \in S\}.$$

A marking M_1 *covers* the marking M_2 if for all $s \in S$, $M_1(s) \geq M_2(s)$. The marking M_2 can be subtracted from M_1 if M_1 covers M_2 and the result of subtraction is the marking; .

$$M_1 - M_2 = \{(s, (M_1(s) + M_2(s))) \mid s \in S, M_1(s) \geq M_2(s)\}.$$

On a given marking M the transition t is *enabled* if M assigns at least one token to each of the input places of t. An enabled transition can fire to give a new marking $M' = ((M - M_{\bullet t}) + M_{\bullet t})$ where

$$M_{\bullet t}(s) = \begin{cases} 1 & \text{if } s \in {}^{\bullet}t \\ 0 & \text{otherwise} \end{cases}$$

$$M_{t^\bullet}(s) = \begin{cases} 1 & \text{if } s \in t^{\bullet} \\ 0 & \text{otherwise} \end{cases}$$

In this case we say M' is *reachable in one step* from M, denoted as $M \to M'$. We say M' is *reachable* from M if there is a sequence of markings $M_1, \ldots, M_n$ such that $M \to M_1 \to \ldots M_n \to M$. The set of all markings reachable from the initial marking M_0 is called the *reachable markings* of the Petri net.

The reachability problem for Petri nets is the following: Given a Petri net $N = (S, T, F, M_0)$ and a marking M, is M reachable from M_0? The problem is known to be decidable [Kos82, May81]. No algorithm for the reachability problem is known that run in elementary time.

The coverability problem for Petri nets is the following: Given a Petri net $N = (S, T, F, M_0)$ and a marking M, is there a reachable marking M' that covers M? This problem is complete for Expspace [Lip76].

2.1.4 Multicounter automata

Definition 2.1.4. *A k-multicounter automaton A is a tuple $A = (Q, \Sigma, \Delta, I, F)$ where Q is a finite set of states, Σ is a finite alphabet, $I \subseteq Q$ is set of initial states and $F \subseteq Q$ is a set of final states. The transition relation is of the form $\Delta \subseteq_{fin} (Q \times \Sigma \times \mathbb{N}^k \times \mathbb{N}^k \times Q)$.*

The automaton works as follows: it has k-counters, denoted by $\bar{v} = (v_1, \ldots v_k)$ which hold non-negative integer values. The automaton starts in an initial state with all its counters empty. During a transition the automaton can increment or decrement some or all of the counters. If a counter holding the value zero is decremented then the automaton halts erroneously.

Formally, a configuration of the machine is of the form $(p, \bar{u})$ where $p \in Q$ and $\bar{u} \in \mathbb{N}^k$. The initial configurations are of the form $(q_0, \bar{0})$, $q_0 \in I$. Given a configuration $(p, \bar{u})$ the automaton can go to a configuration $(q, \bar{v})$ on letter a if there is a transition $(p, a, \bar{v_{dec}}, \bar{v_{inc}}, q) \in \Delta$ such that $\bar{u} - \bar{v_{dec}} \geq \bar{0}$ (pointwise) and $\bar{v} = \bar{u} - \bar{v_{dec}} + \bar{v_{inc}}$. Finally, the automaton accepts if the state reached is final and all the counters are zero. It is known that the non-emptiness problem for multicounter automata and the reachability problem for Petri nets are equivalent in terms of hardness [May81]. Though decidable, it is not known whether the non-emptiness problem for multicounter automata is in elementary time.

We describe a weaker acceptance condition for multicounter automata which is based only on the state of the accepting configuration. With this acceptance condition, the configuration reached is final if the state of the configuration is final. The problem of checking non-emptiness of a multicounter automaton with weak acceptance is known to be EXPSPACE-hard [Lip76].

2.2 Post's Correspondence Problem

Below, we discuss a marvellous tool for showing undecidability results, namely Post's correspondence problem, in short PCP. Let $\Sigma = \{l_1, l_2, \ldots l_k\}$ be a finite alphabet. A PCP instance $I = \{(u_i, v_i) \mid 1 \leq i \leq n, u_i, v_i \in \Sigma^+\}$ is a finite set of ordered pairs of non-empty strings over the alphabet Σ. A solution for I is a finite sequence of integers $i_0, i_1, \ldots i_m$, all of which are from the set $\{1, \ldots n\}$ such that,

$$u_{i_0} u_{i_1} \ldots u_{i_m} = v_{i_0} v_{i_1} \ldots v_{i_m}.$$

The following problem is undecidable: given a PCP instance I, does I have a solution? The problem remains undecidable even when the length of each u_i as well as v_i is at most two [HU79]. We employ this variant also for our proofs.

3

Automata for data words

3.1 Introduction

The theory of finite state automata over (finite) words is an area that is rich in concepts and results, offering interesting connections between computability theory, algebra, logic and complexity theory. Moreover, finite state automata provide an excellent abstraction for many real world applications, such as string matching in lexical analysis [HU79, ASU86], model checking finite state systems[CGP99] etc.

Considering that finite state machines have only bounded memory, it is *a priori* reasonable that their input alphabet is finite. If the input alphabet were infinite, it is hardly clear how such a machine can tell infinitely many elements apart. And yet, there are many good reasons to consider mechanisms that achieve precisely this.

Abstract considerations first: consider the set of all finite sequences of natural numbers (given in binary) separated by hashes. A word of this language, for example, is 100#11#1101#100#10101. Now consider the subset L containing all sequences with some number repeating in it. It is easily seen that L is not regular, it is not even context-free. The problem with L has little to do with the representation of the input sequence. If we were given a bound on the numbers occurring in any sequence, we could easily build a finite state automaton recognizing L. The difficulty arises precisely because we do not have such a bound or because we have 'unbounded data'. It is not difficult to find instances of languages like L occurring naturally in the computing world. For example consider the sequences of all *nonces* used in a security protocol run. Ideally this language should be $\overline{L}$. The question is how to recognize such languages, and whether there is any hope of describing regular collections of this sort.

Note that we could simply take the set of binary numbers as the alphabet in the example above: $\Gamma = \{\#, 0, 1, 10, 11, \ldots\}$. Now, $L = \{w = b_0\#b_1\# \ldots b_n \mid w \in \Gamma^*, \exists i, j.b_i = b_j\}$. Note further that Γ itself is a regular language over the alphabet $\{\#, 0, 1\}$.

There are more concrete considerations that lead to infinite alphabets as well, arising from two strands of computation theory: one from attempts to extend classical model checking techniques to *infinite state systems*, and the other is the realm of *databases*. Systems like software programs, protocols (communication, cryptography, ...), web services and alike are typically infinite state, with many different sources of unbounded data: program data, recursion, parameters, time, communication media, etc. Thus, model checking techniques are confronted with infinite alphabets. In databases, the XML standard format of *semi-structured data* consists of labelled trees whose nodes carry data values. The trees are constrained by schemes describing the tree structure, and restrictions on data values are specified through data constraints. Here again we have either trees or paths in trees whose nodes are labelled by elements of an infinite alphabet.

Building theoretical foundations for studies of such systems leads us to the question of how far we can extend finite state methods and techniques to infinite state systems. The attractiveness of finite state machines can mainly be attributed to the easiness of several decision problems on them. They are robust, in the sense of invariance under nondeterminism, alternation etc. and characterizations by a plurality of formalisms such as Kleene expressions, monadic second order logic, and finite semigroups. Regular languages are logically well behaved (closed under boolean operations, homomorphisms, projections, and so on). What we would like to do is to introduce mechanisms for unbounded data in finite state machines in such a way that we can retain as many of these nice properties as possible.

In the last decade, there have been several answers to this question. We make no attempt at presenting a comprehensive account of all these, but point to some interesting automata theory that has been developed in this direction. Again, while many theorems can be discussed, we concentrate only on one question, that of emptiness checking, guided by concerns of system verification referred to above.

3.2 Languages of data words

Before we consider automaton mechanisms, we discuss languages over infinite alphabets. We will look only at languages of words but it is easily seen that similar notions can be defined for languages of *trees*, whose nodes are labelled from an infinite alphabet. We will use the terminology of database theory, and refer to languages over infinite alphabets as data languages. However, it should be noted that at least in the context of database theory, data trees (as in XML) are more natural than data words, but as it turns out, the questions discussed here happen to be considerably harder for tree languages than for word languages.

Customarily, the infinite alphabet is split into two parts: it is of the form $\Sigma \times \Gamma$, where Σ is a finite set, and Γ is a countably infinite set. Usually, Σ is called the *letter alphabet* and Γ is called the *data alphabet*. Elements of Γ are referred to as *data values*. We use letters a, b etc to denote elements of Σ and use d, d' to denote elements of Γ.

The letter alphabet is a way to provide 'contexts' to the data values. In the case of XML, Σ consists of tags, and Γ consists of data values. Consider the XML description: `<name> ''Tagore''</name>`: the tag `<name>` can occur along with different strings; so also, the string `''Tagore''` can occur as the value associated with different tags. As another example, consider a system of unbounded processes with states $\{b, w\}$ for 'busy' and 'wait'. When we work with the traces of such a system, each observation records the state of a process denoted by its process identifier (a number). A word in this case will be, for example, $(b, d_1)(w, d_2)(w, d_1)(b, d_2)$.

A **data word** w is an element of $(\Sigma \times \Gamma)^*$. A collection of data words $L \subseteq (\Sigma \times \Gamma)^*$ is called a *data language*. In this thesis, by default, we refer to data words simply as words and data languages as languages. As usual, by $|w|$ we denote the length of w.

Let $w = (a_1, d_1)(a_2, d_2) \dots (a_n, d_n)$ be a data word. The *string projection* of w, denoted as $str(w) = a_1 a_2 \dots a_n$, is the projection of w to its Σ components. Let

$L_{\exists n}$	All data words in which at least n distinct data values occur.
$L_{<n}$	All data words in which every data value occurs at most n times.
$L_{a^*b^*}$	All data words whose string projections are in the set a^*b^*.
L_a	All data values under the label a are different.
$L_{a\to b}$	All data values occurring under a occurs under b as well.
L_{dd}	There is a data value which occurs in consecutive positions.

Figure 3.1: Sample data languages

$i \in [n]$. The data class of d_i in w is the set $\{j \in [n] \mid d_i = d_j\}$. A subset of $[n]$ is called a data class of w if it is the data class of some d_i, $i \in [n]$. Note that the set of data classes of w form a partition of $[|w|]$.

We introduce in Figure 3.2 some example data languages which we will keep referring to in the course of our discussion; these are over the alphabet $\Sigma = \{a, b\}, \Gamma = \mathbb{N}$.

Let $\cdot$ denote concatenation on data words. For $L \subseteq (\Sigma\times\Gamma)^*$, consider the Myhill - Nerode equivalence on $(\Sigma\times\Gamma)^*$ induced by L: $w_1 \sim_L w_2$ iff $\forall w \in (\Sigma\times\Gamma)^*, w_1\cdot w \in L \Leftrightarrow w_2 \cdot w \in L$. The language L is said to be regular when $\sim_L$ is of finite index. A classical theorem of automata theory equates the class of regular languages with those recognized by finite state automata, in the context of languages over finite alphabets.

It is easily seen that $\sim_{L_a}$ is not of finite index, since each singleton data word (a, d) is distinguished from (a, d'), for $d \neq d'$. Hence we cannot expect a classical finite state automaton to accept L_a; we need to look for another device, perhaps an infinite state machine.

Indeed, for most data languages, the associated equivalence relation is of infinite index. Is there a notion of *recognizability* that can be defined meaningfully over such languages and yet corresponds (in some way) to finite memory? This is the central question addressed in this and the following chapters.

3.3 Formulating an automaton mechanism

The first challenge in formulating an automaton mechanism is the question of 'finite representability'. It is essential for a machine model that the automaton is presented in a finite fashion. In particular, we need implicit finite representations of the data alphabet. An immediate implication is that we need algorithms that work with such implicit representations. Towards this, it is absolutely necessary that, *we consider only data alphabets* Γ *in which membership and equality are decidable.*

Automata for words over finite alphabets are usually presented as working on a read-only finite tape, with a *tape head* under finite state control. One detail which is often taken for granted is the complexity of the tape head. Since we can recognize a finite language (which is the alphabet!) by a constant-sized circuit the computing power of the tape-head is inferior to that of the automaton.

In the case of infinite alphabets, the situation is different, and our assumption about decidable membership and equality in Γ makes sense when we consider the complexity of the tape head. For example, if we consider the alphabet as the encodings of all halting Turing machines, the tape-head has to be a Σ_1^0 machine, which is obviously hard to conceive of as a machine model relevant to software verification. Therefore, we see that our assumption needs tightening and we should require the membership and equality checking in the alphabet to be *computationally feasible.* In fact, we should also ensure that the language accepted by the automaton, when restricted to a finite subset of the infinite alphabet, remains regular.

One obvious way of implementing finite presentations is by insisting that the finite state automaton uses only *finitely many* data values in its transition relation. However, when the only allowed operation on data is checking for equality of data values, such an assumption becomes drastic: it is easily seen that having infinite data alphabets is superfluous in such automata. Every such machine is equivalent to a finite state machine over a finite alphabet.

Thus we note that infinite alphabets naturally lead us to infinite state systems, whose space of configurations is infinite. The theory of computation is rich in such models: pushdown systems, Petri nets, vector addition systems (VAS), Turing machines etc. In particular, we are led to models in which we equip the automaton with some additional mechanism to enable it to have infinitely many configurations.

This takes us to a striking idea from the 1960's: "automata theory is the study of memory structures". These are structures that allow us to fold infinitely many actions into finitely many instructions for manipulating memory, which can be part of the automaton definition. These are storage mechanisms which provide specific tools for manipulating and accessing data. Obvious memory mechanisms are *registers* (which act like scratch pads, for memorizing specific data values encountered), *stacks*, *queues* etc.

One obvious memory structure is the input tape, which can be 'upgraded' to an unbounded sequential read-write memory. But then it is easily noted that a finite state machine equipped with such a tape is Turing-complete. On the other hand, if the tape is read-only, the machine accepts all data data words whose string projections belong to the letter language (subset of Σ^*) defined by the underlying automaton. Clearly this machine is also not very interesting. We therefore look for structures that keep us in between: those with infinitely many configurations, but for which reachability is yet decidable. Note that such ambition is not unrealistic, since Petri nets and pushdown systems are systems of this kind.

3.4 Register automata

The simplest form of memory is a finite random access read-write storage device, traditionally called *register*. In Register automata [KF94], the machine is equipped with finitely many registers, each of which can be used to store one data value. Every automaton transition includes access to the registers, reading them before the transition and writing to them after the transition. The new state after the transition depends on the current state, the input letter and whether or not the input data value is already stored in any of the registers. If the data value is not stored in any of the registers, the automaton can choose to write it in a register. The transition may also depend on which register contains the encountered data value. The definition we present below is a close variant of the definition in [KF94]. In terms of complexity of decision problems and language acceptance they are equivalent.

Definition 3.4.1. *A* k-**Register automaton** A *is given by* $A = (Q, \Sigma, \Delta, \perp, q_0, F)$, *where* Q *is a finite set of states,* $q_0 \in Q$ *is the initial state,* $F \subseteq Q$ *is the set of final states and* $\perp$ *is the empty register symbol. The transition relation is* $\Delta \subseteq (Q \times \Sigma \times [k] \times Q) \cup (Q \times \Sigma \times Q \times [k])$. *For* $p, q \in Q$, $a \in \Sigma$, $i \in [k]$, *transitions of the form* (p, a, i, q) *are called* read *transitions and transitions of the form* (p, a, q, i) *are called* write *transitions.*

The automaton A has k registers. $\perp$ is used above to denote an uninitialized register. A *configuration* of A is of the form (q, h) where $q \in Q$ denotes the current state and $h : [k] \to (\Gamma \cup \{\perp\})$ is a function from $[k]$ to $(\Gamma \cup \{\perp\})$, such that if for $i \neq j$, $h(i) \in \Gamma$ and $h(j) \in \Gamma$ then $h(i) \neq h(j)$, representing the current register configuration. For convenience, sometimes we identify the function h with the set $\mathsf{range}(h) = \cup_i\{h(i)\}$, for instance by $d \in h$ we mean that the data value d is in the registers. The working of the automaton is as follows. Suppose that A is in state p, with each of the registers i holding data value d_i, and its input is of the form (a, d). Now there are two cases:

- If $d \neq d_i$ for all i, then a register write is enabled and the automaton can make a write transition (p, a, q, i) storing data value d in register i and the next state becomes q.
- Suppose that $d = d_i$, for some $i \in [k]$, and $(p, a, i, q) \in \Delta$. Then this read transition is enabled and when applying the transition, the registers are left untouched and the next state becomes q.

A *run* of A on a data word $w = (a_1, d_1)(a_2, d_2) \ldots (a_n, d_n)$ is a sequence $\gamma = (q_0, h_0)(q_1, h_1) \ldots (q_n, h_n)$, where (q_0, h_0) is the initial configuration of A, and for every $i \in [n]$, there is a transition from (q_{i-1}, h_{i-1}) to (q_i, h_i) on (a_i, d_i) in Δ. γ is *accepting* if $q_n \in F$. The language accepted by A, denoted $L(A) = \{w \in (\Sigma \times \Gamma)^* \mid A \text{ has an accepting run on } w\}$.

Observe that at any configuration all the data values stored in the registers are different.

Example 3.4.2. *Recall the language* L_a *mentioned earlier: it is the set of all data words in which all the data values in context* a *are distinct. The language* $\overline{L_a}$ *can*

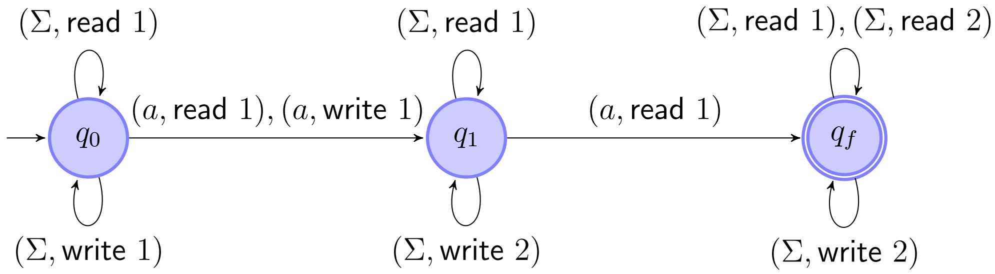

Figure 3.2: Register automaton accepting the language $\overline{L_a}$.

be accepted by a 2-register automaton $A = (Q = \{q_0, q_1, q_f\}, \Sigma, \Delta, \perp, q_0, F = \{q_f\})$, *where* Δ *consists of,*

$$\Delta = \left\{ \begin{array}{c} (q_0, \Sigma, 1, q_0), (q_0, \Sigma, q_0, 1), (q_0, a, 1, q_1), (q_0, \Sigma, q_1, 1), (q_1, \Sigma, 1, q_1), \\ (q_1, \Sigma, q_1, 2), (q_1, a, 1, q_f), (q_f, \Sigma, 1, q_f), (q_f, \Sigma, 2, q_f), (q_f, \Sigma, q_f, 2) \end{array} \right\}$$

A works as follows. Initially A is in state q_0 and stores new input data in the first register. When reading the data value with label a, which appears twice, A changes the state to q_1 nondeterministically and waits there storing the new data in the second register. When the data value stored in the first register appears the second time with label a, A changes state to q_f and continues to be there.

The automaton is shown in the Figure 3.2.

Example 3.4.3. *The language* L_{dd} *is accepted by a* 1*-register automaton* $A = (Q = \{q_0, q_f\}, \Sigma, \Delta, q_0, F = \{q_f\})$ *where*

$$\Delta = \left\{ \begin{array}{c} (q_0, \Sigma, 1, q_f), (q_f, \Sigma, 1, q_f) \\ (q_0, \Sigma, q_0, 1), (q_f, \Sigma, q_f, 1) \end{array} \right\}$$

The automaton (as shown in Figure 3.3) always stores the data values in the register 1 *and stays in state* q_0*, if it sees a data value repeating it goes to state* q_f *and stays there.*

Example 3.4.4. *A finite state automaton is a* 0*-register automaton. Since* a^*b^* *is regular, the language* $L_{a^*b^*}$ *is accepted by a register automaton.*

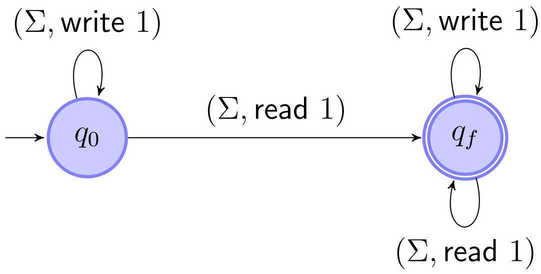

Figure 3.3: 1-Register automaton accepting the language L_{dd}

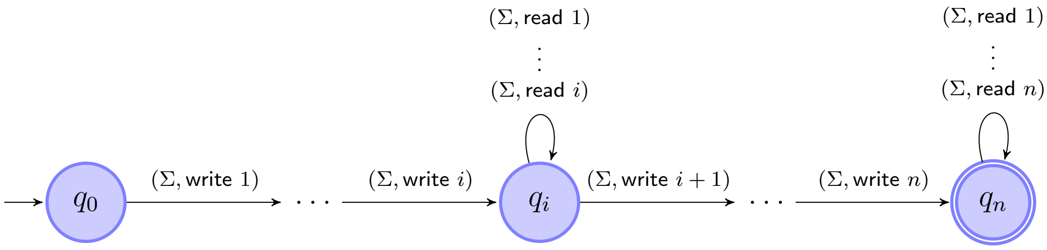

Figure 3.4: n-Register automaton accepting the language $L_{\exists n}$

Example 3.4.5. *The family of languages $L_{\exists n}$ is accepted by n-register automata with $n+1$-states $q_0, \ldots, q_n$ (shown in Figure 3.4) in the following way. The automaton fills up the registers successively with new data values while keeping the number of registers filled in the states. Finally it accepts the word if the state q_n is reached.*

However, the languages $L_{<n}, L_a$ and $L_{a \to b}$ are not accepted by register automata. Below we see why it is so.

Note that a register automaton uses only finitely many registers to deal with infinitely many symbols, and hence we get something analogous to the pumping lemma for regular languages which asserts that a finite state automaton which accepts sufficiently long words also accepts infinitely many words. Suppose there are k registers and the automaton sees $k+1$ data values; since the only places where it can store these data values are in the registers, it is bound to forget one of the data values. This is made precise by the following lemma. Again, our formulation is slightly different from the corresponding lemma in [KF94].

Lemma 3.4.6. *If a k-register automaton A accepts any word at all, then it accepts a word containing at most $k+1$ distinct data values.*

Proof. Let $w = (a_1, d_1)(a_2, d_2)\dots(a_n, d_n)$ be a data word accepted by A and $\rho = (q_0, h_0)(q_1, h_1)\dots(q_n, h_n)$ be an accepting run of A on w. If the size of the set $D = \{d_1, \dots, d_n\}$ is $k+1$ then the claim is proved else let $D' \subset D$ be a subset of size $k+1$. For register configuration h and data values d_1, d_2, we denote by $h[d_1/d_2]$ the register configuration obtained from h by replacing d_1 by d_2. Let,

$$w' = (a_1, d'_1)(a_2, d'_2)\dots(a_n, d'_n)$$

$$\rho' = (q_0, h_0)(q_1, h'_1)\dots(q_n, h'_n), \forall i \geq 1, h'_i = h_i[d_i/d'_i]$$

where $d'_i \in D'$ and $d'_i \in h'_{i-1}$ if and only if $d_i \in h_{i-1}$. We show by induction on n that A has a run ρ' on w' as follows. For the base case, fix $d'_1 = d_1$ and $D' \subseteq D$ such that $d_1 \in D'$ and trivially $(q_0, h_0)\,(q_1, h_1)$ satisfies the conditions. For the inductive step assume that there is a partial sequence $(a_1, d'_1)(a_2, d'_2)\dots\big(a_{i-1}, d'_{i-1}\big)$ and $(q_0, h_0)(q_1, h'_1)\dots(q_{i-1}, h'_{i-1})$ satisfying the above conditions. Assume d_i is stored in register j in h_{i-1}, that is $h_{i-1}(j) = d_i$. We define d'_i to be $h'_{i-1}(j) = d$ and $h'_i = h_i[d_i/d]$. If d_i is not in h_{i-1} then we choose a data value $d \in D'$ not appearing in h'_{i-1} and define $d'_i = d$. Observe that in both these cases the conditions are preserved. However, in order to show that w' is accepted by A, it remains to be proved that $\rho' = (q_0, h'_0)(q_1, h'_1)\dots(q_n, h'_n)$ is an *accepting* run for w'. We prove this using induction again. For the base case it is trivial. For the inductive step, assume d_i is in h_{i-1} in which case there is a read transition (q_{i-1}, a_i, j, q_i) where $h_{i-1}(j) = d_i$. Since d'_i is in h'_{i-1} the same transition is enabled at $\big(q_{i-1}, h'_{i-1}\big)$. Similarly, if d_i is not in h_{i-1} there is a write transition (q_{i-1}, a_i, q_i, j). Since d'_i is also not in h'_{i-1} the same transition is enabled at $\big(q_{i-1}, h'_{i-1}\big)$. This completes the proof. □

Note that the language L_a requires unboundedly many data values to occur with a, and hence by the above lemma, it cannot be recognized by any register automaton. On the other hand, since $\bar{L}_a$ can be accepted by a register automaton, we see that languages recognized by register automata are not closed under complementation. As this suggests, non-deterministic register automata are more powerful than deterministic ones.

While the lemma demonstrates a limitation of register machines in terms of computational power, it also shows the way for algorithms on these machines.

Theorem 3.4.7. *Emptiness checking of register automata is decidable.*

Proof. Let A be a register automaton with k registers, which we want to check for emptiness. Let $D' \subseteq \Gamma, |D'| = k + 1$ be a subset of Γ containing $k + 1$ different values. We claim that $L(A) \neq \emptyset$ if and only if $L(A) \cap (\Sigma \times D')^* \neq \emptyset$. The if direction is trivial. The other direction follows from the preceding lemma. Thus a classical finite state automaton working on a finite alphabet can be employed for checking emptiness of A. □

The emptiness problem for register automata is in NP, since we can guess a word of length polynomial in the size of the automaton and verify that it is accepted. It has also been shown that the problem is complete for NP in [SI00]. The problem is no less hard for the deterministic subclass of these automata. Though, as we mentioned earlier, register automata are not closed under complementation, they are closed under intersection, union, Kleene iteration and homomorphisms.

There are many extensions of the register automaton model. An obvious one is to consider *two-way* machines: interestingly, this adds considerable computational power and the emptiness problem becomes undecidable[NSV04, KF94, Zei06].

The word problem for register automata are NP-complete, while for deterministic register automata it is P-complete[SI00].

3.5 Data and Class Memory automata

The weakness of register automata arises from its finite memory. A way to overcome this is by allowing unbounded memory and hashtables provide an easy mechanism for providing that. Below we discuss an equivalent formulation of such an automaton.

A **Data automaton** [BDM+11] is a composite automaton consisting of a finite state transducer B and a finite state automaton C. They use an internal alphabet Σ' for communication. Formally:

Definition 3.5.1. *A* **data automaton** *is a tuple $A = (B, C)$ where B is a finite state transducer, given by the tuple $B = (Q_b, \Sigma, \Sigma', \Delta_b, O_b, I_b, F_b)$, with input alphabet Σ and output alphabet Σ'. The automaton $C = (Q_c, \Sigma', \Delta_c, I_c, F_c)$ is a finite state auotmaton with alphabet Σ'.*

A run ρ of a data automaton on data word w is defined in the following manner; Let $w' = a_1 \dots a_n$ be the string projection of w. Let $\rho_B = \delta_1 \dots \delta_n \in \Delta_b^*$ be a run of B on w'. The run ρ_B uniquely defines an output word $w'' = O_b(\delta_1) \dots O_b(\delta_n)$ (See section 2.1.2). Let $D(w)$ be the set of data values occurring in w and let w''_d be the subword of w'' formed by the positions labelled by $d \in D(w)$. For each $d \in D(w)$, let ρ_d be a run of the automaton C on w''_d. Define the run ρ as $(\rho_B, \{\rho_d \mid d \in D(w)\})$. We say ρ is successful if (1) ρ_B is a successful run of B on w' (2) For each $d \in D(w)$, ρ_d is a successful run of C on w''_d.

Example 3.5.2. *The language L_a is easily accepted by the following way. The intermediate alphabet is Σ itself. The transducer B is a copy machine, copies every letter to the output. The automaton C accepts the language $\overline{\Sigma^* a \Sigma^* a \Sigma^*}$. It is clear that if in w there is a class with at least two a's then C cannot have a successful run over that class.*

Example 3.5.3. *For accepting the language L_{dd}, choose the intermediate alphabet to be $\{0, 1\}$. While reading the string projection the transducer B chooses two consecutive positions and label them by '1', all other positions are labelled '0'. The automaton C accepts the language $0^*10^*10^* + 0^*$. Note that the automaton C specifies that in each class either all positions are labelled '0' or there are exactly two positions with label '1'. Since the transducer B outputs '1' on two positions, there is atleast one class which contains a '1' and because of C that class contains two '1's. Finally, since B outputs exactly two '1's on consecutive positions it can be inferred that there exist consecutive positions labelled with the same data value.*

Example 3.5.4. *The language $L_{<n}$ is accepted in the following way. Again, the transducer B is a copy machine and the internal alphabet is Σ. The finite state automaton C accepts the language $\Sigma^0 \cup \Sigma^1 \dots \cup \Sigma^n$.*

Example 3.5.5. *In the case of* $L_{a^*b^*}$ *the automaton* B *accepts the language* a^*b^* *and* C *is a machine accepting all strings.*

Example 3.5.6. *For the language* $L_{a\to b}$*, the automaton* B *is a copy machine. Automaton* C *accepts the language* $(a^*ba^*)^*$ *which is the set of strings* w *such that* w *contains a 'b' if it contains an 'a'.*

Now, we give the definition of the finite state automaton equipped with a hash table called Class Memory Automaton (shortly CMA) [BS10].

Definition 3.5.7. *A* **class memory automaton** *is a tuple* $A = (Q, \Sigma, \Delta, q_0, F_\ell, F_g)$ *where* Q *is a finite set of states,* q_0 *is the initial state and* $F_g \subseteq F_\ell \subseteq Q$ *are the sets of* **global** *and* **local** *accepting states respectively. The transition relation is* $\Delta \subseteq (Q \times \Sigma \times (Q \cup \{\perp\}) \times Q)$.

The class memory automaton is equipped with a hashtable h which maps from the set of data values Γ to a finite set of hash values. The working of the automaton is as follows. The finite set of hash values is simply the set of automaton states. A transition of the form (p, a, s, q) on input (a, d) stands for the state transition of the automaton from p to q when the hash value for d is s, as well as the updating of the hash value for d from s to q. The acceptance condition has two parts. The global acceptance set F_g is as usual: after reading the input the automaton state should be in F_g. The local acceptance condition refers to the state of the hash table: the image of the hash function should be contained in F_ℓ. Thus acceptance depends on the memory of the data encountered.

Formally, a hash function is a map $h : \Gamma \to (Q \cup \{\perp\})$ such that $h(d) = \perp$ for all but finitely many data values. h holds the hash value (the state) which is assigned to the data value d when it was read the last time. A configuration of the automaton is of the form (q, h) where h is a hash function. The initial configuration of the automaton is (q_0, h_0) where $h_0(d) = \perp$ for all $d \in \Gamma$.

Transition on configurations is defined as follows: a transition from a configuration (p, h) on input (a, d) to (q, h') is enabled if $(p, a, h(d), q) \in \Delta$, and

$$h'(d') = \begin{cases} q & \text{if} \quad d = d'. \\ h(d') & \text{if} \quad d \neq d'. \end{cases}$$

A *run* of CMA A on a data word $w = (a_1, d_1)(a_2, d_2) \ldots (a_n, d_n)$ is, as usual, a sequence $\gamma = (q_0, h_0)(q_1, h_1) \ldots (q_n, h_n)$, where (q_0, h_0) is the initial configuration of A, and for every $i \in [n]$, there is a transition from (q_{i-1}, h_{i-1}) to (q_i, h_i) on (a_i, d_i) in Δ. γ is *accepting* if $q_n \in F_g$ and for all $d \in \Gamma$, $h_n(d) \in F_l \cup \{\perp\}$. The language accepted by A, denoted $L(A) = \{w \in (\Sigma \times \Gamma)^* \mid A \text{ has an accepting run on } w\}$.

Example 3.5.8. *The language L_a can be accepted by the following class memory automaton $A = (Q, \Sigma, \Delta, q_0, F_l, F_g)$ where $Q = \{q_0, q_a, q_b\}$ and Δ contains the tuples $\{(p, a, \perp, q_a), (p, b, \perp, q_b), (p, b, q_a, q_a), (p, b, q_b, q_b), (p, a, q_b, q_a) \mid p \in \{q_0, q_a, q_b\}\}$. F_l is the set $\{q_a, q_b\}$ and F_g is the set Q. The idea is that for each class the automaton remembers if it has seen an 'a' by means of the hash function. A run terminates erroneously if in a class a second 'a' is seen. Finally the run is successfull if all classes terminates in the local final state q_b.*

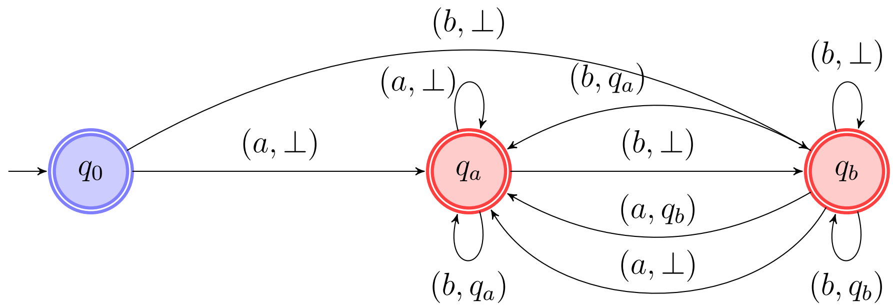

Figure 3.5: CMA accepting the language L_a.

The automaton is shown in the Figure 3.5. The local accepting states are shown in red, while global accepting states are circled.

Example 3.5.9. *The language $L_{a \to b}$ is accepted by CMA in the following fashion (shown in the Figure 3.6). The automaton has three states q_0, q_a, q_b where q_0 is the initial state. For each class if the hash function carries the state q_a then it indicates that so far in the class only 'a' has appeared. Similarly if the hash function indicates q_b then it denotes that the class contains at least one 'b'. The automaton updates the hash function by appropriately changing states on each pair. Finally the automaton accepts if all the classes end in the local final state q_b.*

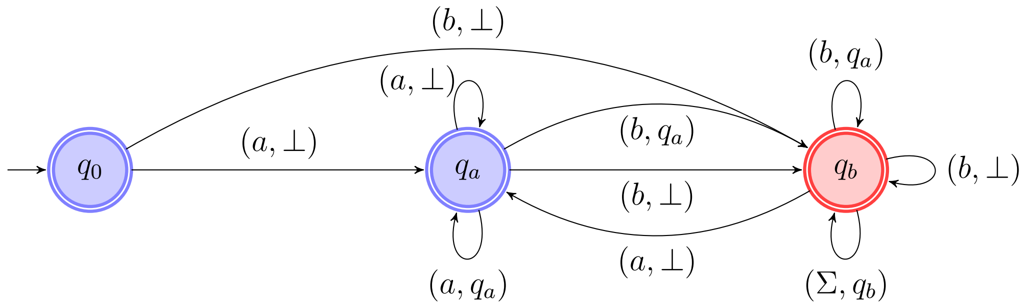

Figure 3.6: CMA accepting the language $L_{a\to b}$.

Example 3.5.10. *The language L_{dd} is accepted by a three state CMA (shown in the Figure 3.7) in the following way. The automaton starts in the initial state q_0. At some point during the run nondeterministically the automaton changes state to q_1. The automaton checks if the following data value is the same by checking the hash function (if it is the case then the state associated with the data value should be q_1) and then moves to the final state q_f. The local final states are irrelevant in this case.*

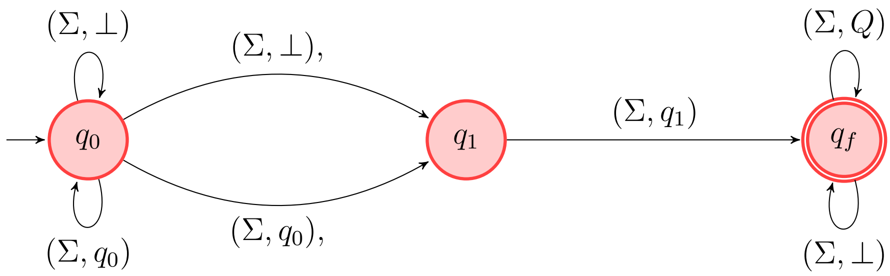

Figure 3.7: CMA accepting the language L_{dd}.

The following two important properties of CMA are proved in [BS10].

Theorem 3.5.11 ([BS10])**.** *CMA and Data automata are expressively equivalent. The translations from CMA to Data automata and vice versa are in* P*.*

Theorem 3.5.12 ([KF94, BS10])**.** *Register automata are strictly less powerful than CMA in terms of expressiveness.*

Next we discuss the emptiness problem for CMA, which follows from the decidability of Data automata. Here we give a proof of the same fact.

Theorem 3.5.13 ([BDM$^+$11, BS10]). *The emptiness problem for CMA is decidable.*

Proof. Let $A = (Q, \Sigma, \Delta, q_0, F_l, F_g)$ be a given CMA. We construct a Petri net N_A and a set of configurations M_A such that A accepts a string if and only if N_A can reach any of M_A.

Define $N_A = (S, T, F)$ where $S = Q \cup \{q^c \mid q \in Q\}$, and the transition relation T is as follows. For each $\delta = (p, a, s, q)$ where $s \neq \perp$ we add a new transition t_δ such that ${}^\bullet t_\delta = \{p, s^c\}$ and $t_\delta^\bullet = \{q, q^c\}$. For each $\delta = (p, a, \perp, q)$ where we add a new transition t_δ such that ${}^\bullet t_\delta = \{p\}$ and $t_\delta^\bullet = \{q, q^c\}$. We add additional transitions $t_{(p,q)}$ for each $p \in F_g, q \in F_l$ such that ${}^\bullet t_{(p,q)} = \{p, q^c\}$ and $t_{(p,q)}^\bullet = \{p\}$. The flow relation is defined accordingly.

The initial marking of the net is M_0 where q_0 has a single token and all other places are empty. M_A is the set of configurations in which exactly one of $q \in F_g$ has a single token and all other places are empty.

The details are routine. The place q^c keeps track of the number of data values with state q. Using induction it can be easily shown that a run of the automaton gives a firing sequence in the net and vice versa. Finally when we reach a global state we can use the additional transitions to pump out all the tokens in the local final states. The only subtlety is that the additional transitions in the net can be used even before reaching an accepting configuration in the net, in which case it amounts to abandoning certain data classes in the run of the automaton (these are data values which are not going to be used again). $\square$

Thus emptiness for CMA is reduced to reachability in Petri nets which is known to be decidable. As it happens, it is also as hard as Petri net reachability[BDM$^+$11]. Since the latter problem is not even known to be elementary, we need to look for subclasses with better complexity. CMA are not closed under complementation, but they are closed under union, intersection, homomorphisms. It also happens that they admit a natural logical characterization to which we will return later.

The word problem for CMA is NP-complete and the complexity remains the same for the deterministic subclass as well [BS10].

3.6 Discussion

In this chapter we saw two popular automaton models for data words, Register and Class Memory automata. While lacking in expressive power register automata have decision problems of relatively low complexity. Class memory automata, on the other hand, have better expressive power but their decision problems are of very high complexity. In the next chapter we discuss a model which falls between these classes both in terms of expressive power and complexity of decision problems.

4

Class counting automata

4.1 Introduction

In this chapter we introduce Class Counting Automata, an extension of finite state automata with counters. We show that the non-emptiness problem for these automata is decidable in elementary time. We also study several extensions of these automata and the complexity of their decision problems. The contents of this chapter appeared in [MR11].

4.2 Class counting automata

A **constraint** $\varphi(x)$ is a univariate inequality of the form $x \leq e$ or $x \geq e$, where $e \in \mathbb{N}$. When $v \in \mathbb{N}$, we say $v \models \varphi(x)$ if $\varphi(v)$ holds. For convenience, often we denote the constraints as $c, c_1, \ldots$. Let C denote the set of all constraints. Define a *bag* to be a finite set $h \subseteq (\Gamma \times \mathbb{N})$ such that whenever $(d, n_1) \in h$ and $(d, n_2) \in h$, we have: $n_1 = n_2$. Thus h defines a partial function from Γ to $\mathbb{N}$ which is defined on a finite subset of Γ. By convention, we implicitly extend it to a total function on Γ by considering h to represent the set $h' = h \cup \{(d, 0) \mid d \notin \mathsf{Domain}(h)\}$. Hence we (ab)use the notation $h(d) = n$ for a bag h. Let $\mathcal{B}$ denote the set of bags. Note that the notation $h \oplus (d, n)$ now stands for the bag $h' = (h - (\{d\} \times \mathbb{N})) \cup \{(d, n)\}$.

The automaton we present below includes a bag of infinitely many monotone counters, one for each possible data value. When it encounters a letter - data pair, say (a, d), the multiplicity of d is checked against a given constraint, and accordingly updated, the transition causing a change of state, as well as possible updates for other data as well. We can think of the bag as a hash table, with elements of Γ as keys, and counters as hash values. Transitions depend only on hash values (subject to constraints) and not keys.

Below, let $\mathsf{Inst} = \{\mathsf{inc}, \mathsf{reset}\}$ stand for the set of *instructions*. We use variables $\pi, \pi_1, \ldots$ to represent the instructions. Each instruction takes a natural number as an argument. The inc instruction with argument k tells the automaton to increment the counter by k, whereas reset with argument k asks for a reset to the value k. Note that the instruction $(\mathsf{inc}, 0)$ says that we do not wish to make any update, and $(\mathsf{inc}, 1)$ causes a unit increment; we use the notation $[0]$ and $[+1]$ for these instructions below.

Definition 4.2.1. *A* **class counting automaton**, *abbreviated as CCA, is a tuple* $CCA = (Q, \Sigma, \Delta, I, F)$, *where* Q *is a finite set of states,* $I \subseteq Q$ *is the set of initial states,* $F \subseteq Q$ *is the set of final states. The transition relation is given by:* $\Delta \subseteq_{fin} (Q \times \Sigma \times C \times \mathsf{Inst} \times \mathbb{N} \times Q)$.

Representation of constants: We note here that the constants in the definition of the automata are represented in unary. The mode of representation of numbers turns out to be crucial for the upper bound of the emptiness problem.

Let A be a CCA. A configuration of A is a pair (q, h), where $q \in Q$ and $h \in \mathcal{B}$. An initial configuration of A is given by (q_0, h_0), where $q_0 \in I$ and h_0 is the empty bag; that is, $\forall d \in \Gamma, h_0(d) = 0$ and $q_0 \in I$.

Given a data word $w = (a_1, d_1), \ldots (a_n, d_n)$, a run of A on w is a sequence $\gamma = (q_0, h_0)(q_1, h_1) \ldots (q_n, h_n)$ such that (q_0, h_0) is an initial configuration and for each $1 \leq i \leq n$ there exists a transition $t_i = (q, a, c, \pi, m, q') \in \Delta$ such that $q = q_i$, $q' = q_{i+1}$, $a = a_{i+1}$ and:

- $h_i(d_{i+1}) \models c$.
- h_{i+1} is given by:

$$h_{i+1} = \begin{cases} h_i \oplus (d_{i+1}, m') & \text{if } \pi = \mathsf{inc}, m' = h_i(d_{i+1}) + m \\ h_i \oplus (d_{i+1}, m) & \text{if } \pi = \mathsf{reset} \end{cases}$$

γ is an accepting run above if $q_n \in F$. The language accepted by A is given by $L(A) = \{w \in (\Sigma \times \Gamma)^* \mid A$ has an accepting run on $w\}$. $L \subseteq (\Sigma \times \Gamma)^*$ is said to be recognizable if there exists a CCA A such that $L = L(A)$. Note that the counters are either incremented or reset to fixed values.

If the configuration $c_2 = (q_2, h_2)$ is reachable from $c_1 = (q_1, h_1)$ on (a, d) we denote it by $c_1 \vdash_{(a,d)} c_2$. Extending this notion further if c_2 is reachable from c_1 on the data word w we denote it by $c_1 \vdash_w c_2$. We first observe that CCA runs have some useful properties. To see this, consider a bag h and $d_1, d_2 \in \Gamma$, $d_1 \neq d_2$ such that at a configuration (q, h), we have two transitions enabled on inputs (a_1, d_1) and (a_2, d_2) leading to configurations (q_1, h_1) and (q_2, h_2) respectively, that is $(q, h) \vdash_{(a_1,d_1)} (q_1, h_1)$ and $(q, h) \vdash_{(a_2,d_2)} (q_2, h_2)$. Notice that for any condition c, if $h(d_2) \models c$ then so also $h_1(d_2) \models c$. Similarly, for any condition c', if $h(d_1) \models c'$ then so also $h_2(d_1) \models c'$. Thus when we have distinct data values, tests on them do not "interfere" with each other. We can extend this observation further: given data words u and v such that the data values in u are pairwise disjoint from those in v, if we have a run from (q, h) on u to (q, h_1) and on v from (q, h_1) to (q', h_2), then there is a configuration (q', h') and a run from (q, h) on v to (q', h'), that is;

$$(q, h) \vdash_u (q, h_1) \vdash_v (q', h_2) \implies \exists h' \in \mathcal{B}, (q, h) \vdash_v (q', h')$$

This observation will be useful in the following.

Example 4.2.2. *The language L_a is accepted by the CCA shown in Figure 4.1. The CCA accepting this language is the automaton $A = (Q, \Sigma, \Delta, \{q_0\}, F)$ where $Q = \{q_0, q_1\}$, q_0 is the only initial state and $F = \{q_0\}$. Δ consists of:*

$$\Delta = \left\{ \begin{array}{c} (q_0, a, x = 0, [+1], q_0), (q_0, a, x = 1, [0], q_1) \\ (q_0, b, x \geq 0, [0], q_0), (q_1, \Sigma, x \geq 0, [0], q_1) \end{array} \right\}$$

The automaton works as follows. Whenever the automaton sees an 'a' it increases the counter corresponding to the data value. On 'b' it does nothing. The automaton moves to a non-final state if it sees an 'a' with the data value whose corresponding counter value is 1.

Since the automaton above is deterministic, by complementing it, that is, set-

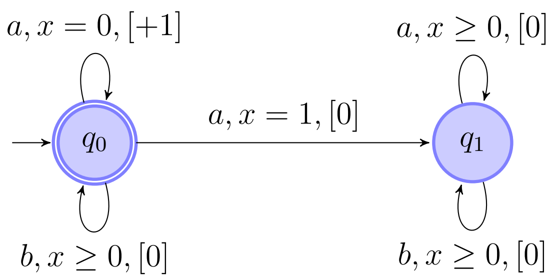

Figure 4.1: CCA accepting the language L_a

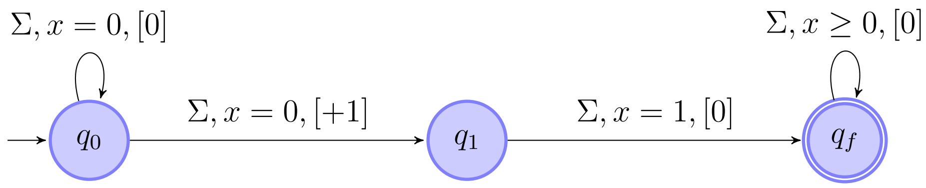

Figure 4.2: CCA accepting the language L_{dd}.

ting $F = \{q_1\}$, we can accept the language $\overline{L_a}$ = "There exists a data value appearing at least twice under a".

Example 4.2.3. *Since a finite state automaton can be viewed as a CCA which does not increase its counters, the language $L_{a^*b^*}$ is recognizable by CCA.*

Example 4.2.4. *The language L_{dd} is accepted by a CCA in the following way (shown in Figure 4.2). The automaton starts in the initial state q_0 with all its counters carrying value 0. Initially the automaton leaves the counters untouched. At some point during the run the automaton nondeterministically increases the counter value to 1 and moves to the state q_1. In the next step the automaton verifies that the counter corresponding to the current data value is 1 and if so the automaton moves to the final state q_f and stays there for the rest of the word.*

Example 4.2.5. *The family of languages $L_{\exists n}$ is accepted by CCA with $(n+1)$-states $q_0, \ldots, q_n$ in the following way (depicted in Figure 4.3). Each fresh data value is marked by increasing the counter corresponding to them and the number of distinct data values is seen is kept in the state. Finally the word is accepted if the number reaches n.*

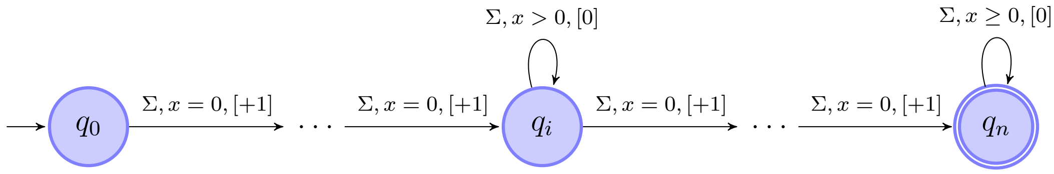

Figure 4.3: CCA accepting the language $L_{\exists n}$

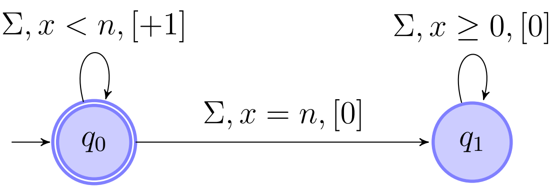

Figure 4.4: CCA accepting the language $L_{<n}$

Example 4.2.6. *The language $L_{<n}$ is accepted by a CCA in the following fashion (shown in Figure 4.4). The automaton starts in the initial state q_0 which is also a final state. During the run the multiplicity of each data value is kept in the counters. If for some data value the multiplicity exceeds n the automaton moves to a non-initial state q_1.*

Example 4.2.7. *Fix Σ to be $\{a\}$. Let the language L_2 be: "There exists a data value whose multiplicity is not two.". The CCA accepting this language is the automaton $A = (Q, \Sigma, \Delta, q_0, F)$ where $Q = \{q_0, q_1, q_2, q_3\}$, q_0 is the only initial state and $F = \{q_1, q_3\}$. Δ consists of:*

$$\Delta = \left\{ \begin{array}{l} (q_0, a, x = 0, [+1], q_1), (q_0, a, x = 0, [0], q_0), (q_1, a, x = 1, [+1], q_2), \\ (q_1, a, x = 0, [0], q_1), (q_2, a, x = 2, [+1], q_3), (q_2, a, x = 0, [0], q_2), \\ (q_3, a, x \geq 0, [0], q_3) \end{array} \right\}$$

The automaton is shown in the Figure 4.5. The idea is that the automaton chooses non-deterministically a data value and faithfully counts its multiplicity, while keeping the counters of other data values zero. Finally the automaton accepts the word, if the current count is not two.

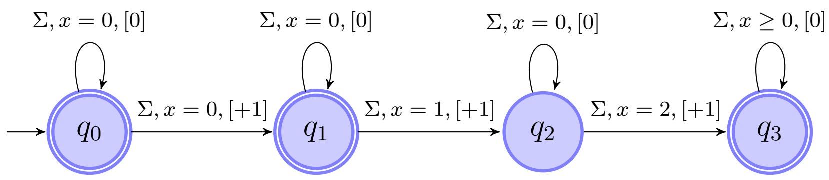

Figure 4.5: CCA accepting the language L_2

But as we show below, its complement language, $\overline{L_2}$ = "All data values occur exactly twice" is not recognizable. Thus, CCA-recognizable data languages are not closed under complementation.

Proposition 4.2.8. *The language $\overline{L_2}$ = "All data values occur exactly twice" is not recognizable.*

Proof. Suppose there is a CCA A with m states accepting this language. Consider the data word

$$w = (a, d_1)(a, d_2) \dots (a, d_{m+1})(a, d_1)(a, d_2) \dots (a, d_{m+1})$$

Clearly, $w \in \overline{L_2}$. Therefore, there is a successful run of A on w. Then there is a state q repeating in the suffix of length $m + 1$. Let us say this splits w as $u \cdot v \cdot v'$, such that the configuration after u is (q, h) and after v it is (q, h_1). Then by the remarks we made earlier, we can find an accepting run for $u \cdot v'$ as well. But then $u \cdot v'$ is not in $\overline{L_2}$. $\square$

Proposition 4.2.9. *CCA-recognizable data languages are closed under union and intersection but not under complementation.*

Proof. Closure under union is easily obtained by non-determinism. Closure under intersection requires the use of more than one bag which we will discuss later. $\square$

The following observation will be useful for decision questions that follow. Given a CCA $A = (Q, \Sigma, \Delta, I, F)$ let m be the maximum constant used in Δ. We define the following equivalence relation on $\mathbb{N}$, $e \simeq_{m+1} e', e, e' \in \mathbb{N}$ iff $e < (m + 1) \vee e' < (m + 1) \Rightarrow e = e'$. Note that if $e \simeq_{m+1} e'$ then a transition is enabled at e if and only if it is enabled at e'. We can extend this equivalence to configurations of the CCA as follows. Let $(q_1, h_1) \simeq_{m+1} (q_2, h_2)$ iff $q_1 = q_2$ and $\forall d \in \Gamma, h_1(d) \simeq_{m+1} h_2(d)$.

Lemma 4.2.10. *If c_1, c_2 are two configurations of the CCA such that $c_1 \simeq_{m+1} c_2$, then $\forall w \in (\Sigma \times \Gamma)^*$, $c_1 \vdash_w c_1' \implies \exists c_2', c_2 \vdash_w c_2'$ and $c_1' \simeq_{m+1} c_2'$.*

Proof. Proof by induction on the length of w. For the base case observe that any transition enabled at c_1 is enabled at c_2 and the counter updates respects the equivalence. For the inductive case consider the word $w \cdot (a, d)$. By induction hypothesis $c_1 \vdash_w c_1' \implies \exists c_2', c_2 \vdash_w c_2'$ and $c_1' \simeq_{m+1} c_2'$. If $c_1' \vdash_{(a,d)} c_1''$ then using the above argument there exists c_2'' such that $c_2' \vdash_{(a,d)} c_2''$ and $c_1'' \simeq_{m+1} c_2''$. □

In fact the lemma holds for any $N \geq m+1$, where m is the maximum constant used in Δ. This observation paves the way for proving the decidability of the emptiness problem.

4.3 Decision problems

Since the space of configurations of a CCA is infinite, reachability is in general non-trivial to decide. We now show that the emptiness problem is elementarily decidable.

Theorem 4.3.1. *The non-emptiness problem for CCA is* EXPSPACE*-complete.*

4.3.1 Upper bound

We reduce the emptiness problem of CCA to the covering problem on Petri nets ([Esp96]). For checking emptiness, we can omit the Σ labels from the configuration graph; we are then left only with counter behavior. However since we have unboundedly many counters, we are led to the realm of multi-counter automata, or vector addition systems.

Definition 4.3.2. *An ω-counter machine B is a tuple (Q, Δ, I) where Q is a finite set of states, $I \subseteq Q$ is the set of initial states and $\Delta \subseteq_{fin} (Q \times C \times \mathsf{Inst} \times \mathbb{N} \times Q)$.*

A configuration of B is a pair (q, h), where $q \in Q$ and $h : \mathbb{N} \to \mathbb{N}$. The initial configurations of B are of the form (q_0, h_0) where $q_0 \in I$ and $h_0(i) = 0$ for all i in $\mathbb{N}$. A run of B is a sequence $\gamma = (q_0, h_0)(q_1, h_1) \ldots (q_n, h_n)$ such that for all i such that $0 \leq i < n$, there exists a transition $t_i = (p, c, \pi, m, q) \in \Delta$ such that $p = q_i$, $q = q_{i+1}$ and there exists j such that $h(j) \models c$, and the counters are updated in a similar fashion to that of CCA.

The reachability problem for B asks, given $q \in Q$, whether there exists a run of B from (q_0, h_0) ending in (q, h) for some h ("Can B reach q?").

Lemma 4.3.3. *Checking emptiness for CCA can be reduced to checking reachability for ω-counter machines.*

Proof. It suffices to show, given a CCA, $A = (Q, \Sigma, \Delta, I, F)$, where $F = \{q\}$, that there exists a counter machine $B_A = (Q, \Delta', I)$ such that A has an accepting run on some data word exactly when B_A can reach q. (When F is not singleton, we simply repeat the construction.) Δ' is obtained from Δ by converting every transition (p, a, c, π, m, q) to (p, c, π, m, q). Now, let $L(A) \neq \emptyset$. Then there exists a data word w and an accepting run $\gamma = (q_0, h_0)(q_1, h_1) \ldots (q_n, h_n)$ of A on w, with $q_n = q$. Let $g : \mathbb{N} \to \Gamma$ be an enumeration of data values. It is easy to see that $\gamma' = (q_1, h_0 \circ g)(q_1, h_1 \circ g) \ldots (q_n, h_n \circ g)$ is a run of B_A reaching q.

($\Leftarrow$) Suppose that B_A has a run $\eta = (q_0, h_0)(q_1, h_1) \ldots (q_n, h_n)$, $q_n = q$. It can be seen that $\eta' = (q_0, h_0 \circ g^{-1})(q_1, h_1 \circ g^{-1}) \ldots (q_n, h_n \circ g^{-1})$ is an accepting run of A on $w = (a_1, d_1) \ldots (a_n, d_n)$ where w satisfies the following. Let (p, c, π, m, q) be the transition of B_A taken in the configuration (q_i, h_i), and d_k such that $h_i(d_k) \models c$. Then by the definition of B_A there exists a transition (p, a, c, π, m, q) in Δ. Then it should be the case that $a_{i+1} = a$ and $d_{i+1} = g(d_k)$. $\square$

Proposition 4.3.4. *Checking non-emptiness of ω-counter machines is decidable.*

Let $s \subseteq \mathbb{N}$, and c a constraint. We say $s \models c$, if for all $n \in s$, $n \models c$.

We define the following partial function Bnd on all finite and co-finite subsets of $\mathbb{N}$. Given $s \subseteq_{fin} \mathbb{N}$, $Bnd(s)$ is defined to be the least number greater than all the elements in s. If s is a co-finite subset of $\mathbb{N}$, $Bnd(s)$ is defined to be $Bnd(\mathbb{N} \backslash s)$.

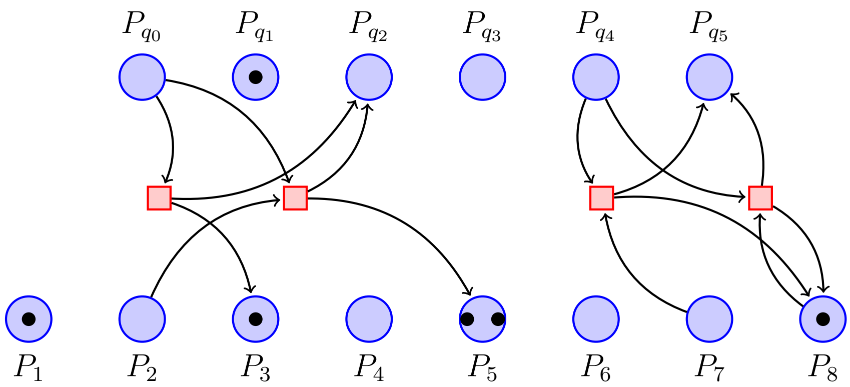

Figure 4.6: Transitions corresponding to $(q_0, x < \mathbf{1}, \mathsf{inc}, 3, q_2)$, $(q_0, x = \mathbf{2}, \mathsf{inc}, 3, q_2)$ and $(q_4, x \geq \mathbf{6}, \mathsf{inc}, 1, q_5)$.

Proof.

Given an ω-counter machine $B = (Q, \Delta, q_0)$ let

$$m_B = max\{Bnd(s) \mid s \models c, c \text{ is used in } \Delta\}.$$

It is worth noting that m_B is of size $\mathcal{O}(|A|)$.

We construct a Petri net $N_B = (S, T, F, M_0)$ where,

- $S = \{P_q \mid q \in Q\} \cup \{P_i \mid i \in \mathbb{N}, 1 \leq i \leq m_B\}$.

- T is defined according to Δ as follows. Let $(p, c, \pi, n, q) \in \Delta$ and let i be such that $0 \leq i \leq m_B$ and $i \models c$. Then we add a transition t such that ${}^\bullet t = \{P_p, P_i\}$ and $t^\bullet = \{P_q, P_{i'}\}$, where (i) if π is inc then $i' = min\{m_B, i+n\}$, and (ii) if π is reset then $i' = min\{m_B, n\}$. Note that i can be zero, in which case we add edges only for the places in $\{P_i \mid i \in [m_B]\}$.

  Formally we define T as follows. Given a transition $\delta = (p, c, \pi, n, q) \in \Delta$, let $I(\delta) \subseteq (\{0, 1, \ldots, m_B\} \times \{0, 1, \ldots, m_B\})$ be the pairs (i, i') such that,

$$I(\delta) = \left\{ \begin{array}{l} (i, i') \mid i \models c, \pi = \mathsf{inc}, i' = min\{m_B, i+n\} \\ (i, i') \mid i \models c, \pi = \mathsf{reset}, i' = min\{m_B, n\} \end{array} \right\}$$

Finally, T is defined as,

$$T = \bigcup_{\delta=(p,c,\pi,n,q)\in\Delta} \left\{ \begin{array}{l} (\{P_p, P_i\}, \{P_q, P_{i'}\}) \mid i \neq 0, i' \neq 0, (i,i') \in I(\delta) \\ (\{P_p\}, \{P_q, P_{i'}\}) \mid i' \neq 0, (0,i') \in I(\delta) \\ (\{P_p, P_i\}, \{P_q\}) \mid i \neq 0, (i,0) \in I(\delta) \\ (\{P_p\}, \{P_q\}) \mid (0,0) \in I(\delta) \end{array} \right\}$$

- The flow relation F is defined according to ${}^\bullet t$ and $t^\bullet$ for each $t \in T$.
- The initial marking is defined as follows. $M_0(P_{q_0}) = 1$ and for all p in S, if $p \neq q_0$ then $M_0(P_p) = 0$.

Let M be any marking of N_B. We say that M is a *state marking* if there exists $q \in Q$ such that $M(P_q) = 1$ and $\forall p \in Q$ such that $p \neq q$, $M(P_p) = 0$. When M is a state marking, and $M(P_q) = 1$, we speak of q as the state marked by M. For $q \in Q$, define $M_f(P_q)$ to be set of state markings that mark q. It can be shown, from the construction of N_B, that in any reachable marking M of N_B, if there exists $q \in Q$ such that $M(P_q) > 0$, then M is a state marking, and q is the state marked by M.

We now show that the counter machine B can reach a state q iff N_B has a reachable marking which covers a marking in $M_f(P_q)$. We define the following equivalence relation on $\mathbb{N}$, $m \simeq_{m_B} n$ iff $(m < m_B) \vee (n < m_B) \Rightarrow m = n$. We can lift this to the bags (in ω-counters) in the natural way: $h \simeq_{m_B} h'$ iff $\forall i\, (h(i) < m_B) \vee (h'(i) < m_B) \Rightarrow h(i) = h'(i)$. It can be easily shown that if $h \simeq_{m_B} h'$ then a transition is enabled at h if and only if it is enabled at h'.

Let μ be a mapping of B-configurations to N_B-configurations as follows: given $\chi = (q, h)$, define $\mu(\chi) = M_\chi$, where

$$M_\chi(P_p) = \left\{ \begin{array}{ll} 1 & \text{iff } p = q \\ 0 & \text{iff } p \in Q \backslash \{q\} \\ |[i]| & \text{iff } P_p = P_i \end{array} \right\}$$

Above $[i]$ denotes the equivalence class of i under $\simeq_{m_B}$ on $\mathbb{N}$ in h. Now suppose that B reaches q. Let the resulting configuration be $\chi = (q, h)$. We claim that the marking $\mu(\chi)$ of N_B is reachable (from M_0) and covers $M_f(P_q)$. Conversely if a reachable marking M of N_B covers $M_f(P_q)$, for some $q \in Q$, then there exists a reachable configuration $\chi = (q, h)$ of B such that $\mu(\chi) = M$.

From the claim it follows that checking reachability of q in B reduces to checking reachability of a marking which covers M such that $M(P_q) = 1$ and for all other places p, $M(p) = 0$.

($\Rightarrow$) The proof is by induction on the length of the B-run. For the base case, observe that $\mu(\chi_0) = M_0$, which is a state marking that marks q_0. Assume that for every run of length n the claim is true.

Suppose that $\chi = (q, f)$ is a configuration reachable in n steps, and that the transition $t = (q, c, \pi, m, q')$ can be taken at χ on counter i such that $f(i) \models c$, resulting in the configuration $\chi' = (q', f')$. By induction hypothesis there exists a marking M such that $\mu(\chi) = M$. By definition of μ it is the case that $M(P_q) = 1$.

If $f(i) = 0$ then the transition $t_0 \in T$ with ${}^\bullet t_0 = \{P_q\}$ is enabled (since its only input place, namely P_q contains a token) and is fired. In the resulting marking M', if $q \neq q'$ then $M'(P_q) = 0$ and $M'(P_{q'}) = 1$, else $M'(P_q) = M(P_q)$ since $P_{q'} \in t_0^\bullet$. If $f(i)$ is updated to $f'(i) = 0$ then $t_0^\bullet = \{P_{q'}\}$, which means the transition t did not increment the counter i or reset it to zero. In which case for all $u \in [m_B]$ it is the case that $M'(u) = M(u)$. Hence $\mu(\chi') = M'$. If $f(i)$ is updated to $f'(i) > 0$ then $t_0^\bullet = \{P_{q'}, P_{v'}\}$ where $v' \simeq_{m_B} f'(i)$, in which case, $M'(P_{v'}) = M(P_{v'}) + 1$ and for all $u \in [m_B] \backslash \{v'\}$ is the case that $M'(u) = M(u)$. Hence again $\mu(\chi') = M'$.

If $f(i) > 0$ then there exists $v \in [m_B]$ such that $M(P_v) > 0$ and $v \simeq_{m_B} f(i)$. Then $t_v \in T$ with ${}^\bullet t_v = \{P_q, P_v\}$ is enabled and is fired. Again, in the resulting marking M', if $q \neq q'$ then $M'(P_q) = 0$ and $M'(P_{q'}) = 1$, else $M'(P_q) = M(P_q)$, since $P_{q'} \in t_v^\bullet$. If $f(i)$ is updated to $f'(i) = 0$ then $M'(P_v) = M(P_v) - 1$ and for all $u \in [m_B] \backslash \{v\}$ it is the case that $M'(u) = M(u)$. Hence $\mu(\chi') = M'$. If $f(i)$ is updated to $f'(i) > 0$ then $t_0^\bullet = \{P_{q'}, P_{v'}\}$ where $v' \simeq_{m_B} f'(i)$. In which case, if $v \neq v'$ then $M'(P_v) = M(P_v) - 1$, $M'(P_{v'}) = M(P_{v'}) + 1$ and for all $u \in [m_B] \backslash \{v, v'\}$ it is the case that $M'(u) = M(u)$. If $v = v'$ then for all $u \in [m_B]$ it is the case that $M'(u) = M(u)$. Hence again, $\mu(\chi') = M'$.

Thus $\mu(\chi')$ is reachable from M in one step by firing t'.

($\Leftarrow$) The proof in the other direction is similar. We do induction on the length of the N_B-marking sequence. For the base case, as in the previous case $\mu(\chi_0) = M_0$. Assume that for every marking sequence of length n the claim is true.

We are considering only one case below; other cases follow similarly. Suppose that M is a marking reachable in n steps, and that the transition $t_v = (\{P_q, P_v\}, \{P_{q'}, P_{v'}\}), q, q' \in Q, v, v' \in [m_B]$ is enabled at M and is fired resulting in the marking M'. By induction hypothesis there exists a B-configuration $\chi = (q, f)$ such that $\mu(\chi) = M$. There exists an $i \in \mathbb{N}$ such that $f(i) \simeq_{m_B} v$ since $M(P_v) > 0$. By construction, the transition t_v was formed from a transition $t = (q, c, \pi, m, q'), t \in \Delta$ in B such that $v \models c$ and therefore $f(i) \models c$. Therefore the transition can be taken in B resulting in configuration $\chi' = (q', f')$ such that updating $f(i)$ with respect to π and m will result in a value $f'(i)$ which is m_B-equivalent to v'. This is by virtue of the construction of t_v. Hence, $\mu(\chi') = M'$.

Since the covering problem for Petri nets is decidable, so is reachability for ω-counter machines and hence emptiness checking for CCA is decidable. □

Complexity of Emptiness checking: The decision procedure discussed above runs in EXPSPACE[Esp96], and thus we have elementary decidability. Note that the representation of constants in unary is a crucial assumption about the EXPSPACE upper bound. When the constants are represented in binary, we do not know whether the upper bound still holds.

4.3.2 Lower bound

We now show that the emptiness problem is also EXPSPACE-hard. Effectively this is a reduction of the covering problem again, but for technical convenience, we use multicounter automata.

A k-multicounter automaton with weak acceptance is a tuple $A = (Q, \Sigma, \Delta, q_0, F)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is a set of final states. The transition relation is of the form $\Delta \subseteq_{fin} (Q \times \Sigma \times \mathbb{N}^k \times \mathbb{N}^k \times Q)$. The two vectors in the transition specify decrements and increments of the counters.

The automaton works as follows: it has k-counters, denoted by $\bar{v} = (v_1, \dots v_k)$ which hold non-negative counter values. A configuration of the machine is of the form $(q, \bar{v})$ where $q \in Q$ and $\bar{v} \in \mathbb{N}^k$. The initial configuration is $(q_0, \bar{0})$. Given a configuration $(q, \bar{v})$ the automaton can go to a configuration $(q', \bar{v}')$ on letter a if there is a transition $(q, a, \bar{v_{dec}}, \bar{v_{inc}}, q')$ such that $\bar{v} - \bar{v_{dec}} \geq \bar{0}$ (pointwise) and $\bar{v}' = \bar{v} - \bar{v_{dec}} + \bar{v_{inc}}$. A final configuration is one in which the state is final.

The problem of checking non-emptiness of a multicounter automaton with weak acceptance is known to be EXPSPACE-hard [Lip76].

Any multicounter automaton $M = (Q, \Sigma, \Delta, q_0, F)$ can be converted to another (in a "normal form"): $M' = (Q', \Sigma, \Delta', q_0, F)$ such that $L(M)$ is non-empty if and only if $L(M')$ is non-empty and M' uses only unit vectors or zero vectors in its transitions. A unit vector is of the form $(b_1, b_2, \ldots, b_k)$ where there is a unique $i \in [k]$ such that $b_i = 1$ and for $j \neq i$, $b_j = 0$. That is M' decrements or increments at most one counter in each transition.

Δ' is obtained as follows. Let $t = (q, a, \bar{v_{dec}}, \bar{v_{inc}}, q')$. Let $\bar{u_1}, \bar{u_2}, \ldots, \bar{u_n}$ be a sequence of unit vectors such that $\bar{v_{dec}} = \Sigma_i \bar{u_i}$ and $\bar{u_1}', \bar{u_2}', \ldots, \bar{u_m}'$ be a sequence of unit vectors such that $\bar{v_{inc}} = \Sigma_i \bar{u_i}'$. We add intermediate states to rewrite t by the following sequence of transitions,

$$(q, a, \bar{u_1}, \bar{0}, q_{(t,\bar{u_1})}), (q_{(t,\bar{u_1})}, a, \bar{u_2}, \bar{0}, q_{(t,\bar{u_2})}), \ldots, (q_{(t,\bar{u_n})}, a, \bar{0}, \bar{u_1}', q_{(t,\bar{u_1}')}),$$

$$(q_{(t,\bar{u_1}')}, a, \bar{0}, \bar{u_2}', q_{(t,\bar{u_2}')}), \ldots, (q_{(t,\bar{u_{m-1}}')}, a, \bar{0}, \bar{u_m}', q')$$

Lemma 4.3.5. *$L(M)$ is non-empty if and only if $L(M')$ is non-empty.*

Proof. By an easy induction on the length of the run. It is easy to see that for every accepting run ρ of M we have an accepting run ρ' of M', this is achieved by replacing every transition t in the run ρ by the corresponding sequence of transitions. For the reverse direction, we need to show that every run accepting run ρ' of M' can be translated to an accepting run ρ of M. This is possible since the intermediate states added to obtain the transitions in M' are unique for each transition t in M. Hence for every sequence of transitions taking M' from q_1 to q_2 where $q_1, q_2 \in Q$ there is a unique transition t which takes M from q_1 to q_2. By doing an induction on the number of states occurring in ρ' which are from Q we can show that there is a valid run ρ which is accepting. $\square$

Next we convert M' to a CCA thus establishing a lower bound of EXPSPACE for the emptiness problem. Let $M' = (Q, \Sigma, \Delta, q_0, F)$ be a k-multicounter automaton in normal form. We construct the automaton $A = (Q, \Sigma, \Delta_A, q_0, F)$. Let $t = (q, a, \bar{u}, \bar{u}', q')$ where $\bar{u}, \bar{u}'$ are either unit or zero vectors. If $\bar{u}$ is the i-th unit vector and $\bar{u}'$ is a zero vector, we add a transition $t_A = (q, a, (x = i), (\mathsf{reset}, 0), q')$ to Δ_A. If $\bar{u}$ is the i-th unit vector and $\bar{u}'$ is the j-th unit vector, we add a transition $t_A = (q, a, (x = i), (\mathsf{reset}, j), q')$ to Δ_A. If $\bar{u}$ is a zero vector and $\bar{u}'$ is the j-th unit vector, we add a transition $t_A = (q, a, (x = 0), (\mathsf{reset}, j), q')$ to Δ_A.

Lemma 4.3.6. *$L(M')$ is non-empty if and only if $L(A)$ is non-empty.*

Proof. The proof is by induction on the length of the run. First we define a mapping from configurations of A to configurations of M' in the following manner, $\mu((q, \bar{h})) = (q, \bar{v})$ where $v_i = |\{j \mid \bar{h}(j) = i\}|$. We show, by induction on the length of the run, that for every configuration χ reachable by A there is a configuration ψ of M' such that $\mu(\chi) = \psi$ and conversely for every configuration ψ reachable by M' there is a configuration χ reachable by A such that $\mu(\chi) = \psi$.

For the base case, it is evident that $\mu((q_0, \bar{h_0})) = (q_0, \bar{0})$.

Suppose that $\chi = (q, \bar{h})$ is a configuration reachable in l steps, and that the transition $t = (q, a, x = j, (\mathsf{reset}, i), q')$ is enabled at χ. Therefore there is a counter holding the value j. By induction hypothesis there exists a configuration ψ such that $\mu(\chi) = \psi = (q, \bar{v})$ such that $v_j > 0$. After the transition t, the number of counters holding the value j decreases by one and the number of counters holding the value i increases by one (if $i \neq 0$). This is achieved by the transition $(q, a, \bar{u_j}, \bar{u_i}, q')$ in Δ', preserving the map μ.

Conversely, suppose a configuration $\psi = (q, \bar{v})$ is reachable by M' in l steps. Then by induction hypothesis we have a configuration χ reachable by the automaton A such that $\mu(\chi) = \psi$. Suppose a transition $t' = (q, a, \bar{u_i}, \bar{u_j}, q')$ is enabled in ψ resulting in ψ'.

Consider the case where $\bar{u_i} \neq \bar{0}$ and $\bar{u_j} \neq \bar{0}$. By construction t' is obtained from a transition $t = (q, a, (x = i), \mathsf{reset}, j, q')$. We choose the smallest counter holding the value zero and apply the transition t, resulting in ξ' such that $\mu(\xi') = \psi'$. The remaining cases are similar. ☐

The reduction from M to M' is not in polynomial time when the constants in the transitions of the Multicounter automata are encoded in binary. However, we observe that the EXPSPACE-hardness for covering problem from [Esp96, Lip76] can be obtained with updates restricted to the values -1, 0 and 1. Hence, the lower bound extends to the scenario where the constants are represented in binary.

4.3.3 Word problem

Since emptiness checking is of such high complexity, one may wonder whether the model is complex enough to render even the word problem to be hard: the simplest algorithmic question of how one can check whether a given word is accepted or not. The important thing to note is that during a run, the size of the configuration is bounded by the length of the input data word. Therefore a non-deterministic Turing machine can easily guess a path in polynomial time and check for acceptance. Hence the word problem is easily seen to be in NP. Interestingly, it turns out to be NP-hard as well.

Theorem 4.3.7. *The word problem for CCA is NP-complete.*

Proof. The proof is by reduction of the satisfiability problem for 3-CNF formulas to the word problem for CCAs. Given the 3-CNF formula, we code it up as a data word, where data values are used to remember the identity of literals in clauses. We use a two letter alphabet with $+, -$ indicating whether a propositional variable occurs positively or negatively. Data values stand for the propositional variables themselves. Thus a pair $(+, d_1)$ asserts that the first boolean variable occurs positively.

We show the coding by an example, let $\varphi \equiv (p_1 \vee \neg p_3 \vee p_4) \wedge (\neg p_2 \vee p_5 \vee p_1) \wedge (\neg p_3 \vee \neg p_4 \vee p_5)$, we construct the corresponding word over the alphabet $\{+, -, \#\} \times \Gamma$,

$$w = (+, d_1)(-, d_3)(+, d_4)(\#, d)(-, d_2)(+, d_5)(+, d_1)(\#, d)(-, d_3)(-, d_4)(+, d_5)(\#, d)$$

The non-deterministic automaton checks satisfiability in the following way. Every time the automaton encounters a new data value (representing a propositional variable), the automaton non-deterministically assigns a boolean value and stores it in the counter (1 for $\perp$ and 2 for $\top$) corresponding to the data value, in the future whenever the same data value occurs the counter is consulted to obtain the assigned value to the propositional variable. The automaton evaluates each clause and carries the partial evaluation in its state. Finally the automaton accepts the word if the formula evaluates to $\top$. □

4.4 Extensions and subclasses

We observe that the model admits many extensions, without substantially affecting the main decidability result.

4.4.1 Deterministic CCA

To define the deterministic subclass of CCA, we need a way of ensuring that nondeterminism is only on Q. Towards this, we say that two constraints c_1 and c_2 are *non-intersecting* if there does not exist $v \in \mathbb{N}$ such that $v \models c_1$ and $v \models c_2$. Observe that any automaton can be converted to an automaton in which the transitions are such that:

- If $(q, a, c_1, \pi_1, m_1, q_1) \in \Delta$, $(q, a, c_2, \pi_2, m_2, q_2) \in \Delta$ and $c_1 \neq c_2$, then c_1 and c_2 are non-intersecting.

An automaton A is a *deterministic* class counting automaton (DCCA) if it is a CCA with the property mentioned above and whenever $(q, a, c, \pi_1, m_1, q_1) \in \Delta$ and $(q, a, c, \pi_2, m_2, q_2) \in \Delta$, we have $\pi_1 = \pi_2, m_1 = m_2$ and $q_1 = q_2$. Since the size of the configuration is bounded by the size of the data word, the word problem of DCCA is in P. Also by an easy reduction from Monotone-CVP we can show that the problem is P-hard.

Proposition 4.4.1. *The word problem for DCCA is P-complete.*

Proof. It is easy to see that the size of the configuration of an automaton on a word is bounded by the length of the word. Hence checking membership is polynomial time in the length of the word, hence in P. For completeness we reduce the circuit valuation problem (CVP) to the membership problem of a CCA. Circuit valuation problem asks the following question; Given a circuit C and a valuation V, does the circuit evaluate to $\top$? We assume that the circuit is presented in a topologically sorted order. For example, let the circuit be

$$c_0 = p_0 \vee \neg p_1, c_1 = \neg p_0 \wedge p_2, C = c_0 \vee c_1$$

and the valuation be $(p_0, 0), (p_1, 1), (p_2, 1)$. We construct a word w coding both the circuit and the evaluation in the following way,

$$w = (\bot, d_0)(\top, d_1)(\top, d_2)(;, d)(+, d_0)(-, d_1)(\vee, c_0)(-, d_0)(+, d_2)(\wedge, c_1)(+, c_0)(+, c_1)(\vee, C)$$

Here the data values $d_0, d_1, \ldots$ stand for the input variables and $c_0, c_1, \ldots$ represent the gates. The automaton works in two phases. In the first phase, before encountering the letter $';'$ the automaton consults the letters from $\{\top, \bot\}$ to initialize the counter corresponding to the data values to either 1 (for $\bot$) or 2 (for $\top$). Once the automaton reaches the letter $';'$ it moves on to the evaluation phase where it evaluates each gate and stores the output value of the gate in the counter corresponding to the data value denoting the gate. Computing the output value of a gate depends on the value of the input values (appropriated with their signs, $+$ or $-$) and the type of gate ($\vee$ or $\wedge$. Finally the automaton accepts if the last gate has value $\top$. □

The restriction of determinism makes DCCA strictly weaker than CCA as shown by the following proposition.

Proposition 4.4.2. *The language L_{dd} is not accepted by any DCCA.*

Proof. The proof is by contradiction. Assume L_{dd} is accepted by a DCCA with m states. Consider the data word $w = (a, d_1)(a, d_2) \ldots (a, d_n)$ such that all data values are distinct and $n = 2 \cdot m + 1$. Let $C_0, C_1, C_2 \ldots C_n$ be the unique run of the automaton on w, where $C_i = (q_i, \bar{h_i})$. By pigeonhole principle there are two configurations C_i and C_j, $1 \leq i < j \leq n$, such that $q_i = q_j$ and $\bar{h_i}(d_i) = \bar{h_j}(d_j)$. Let $w\restriction_i = (a, d_1)(a, d_2) \ldots (a, d_i)$ be the prefix of w of length i. Since $w\restriction_j \cdot (a, d_j) \in L_{dd}$, there is a transition t enabled at C_j on (a, d_j) such that $C_j \vdash_t C_f$, where C_f is a final configuration. Since $\bar{h_i}(d_i) = \bar{h_j}(d_j)$ and all data values are distinct, t is enabled at C_j on (a, d_i) also. Therefore the automaton accepts $w\restriction_j \cdot (a, d_i)$ as well, though it is not in the language. $\square$

Recall that L_{dd} on the other hand is accepted by a register automaton. This along with the fact that L_a is accepted by a DCCA (which is not accepted by register automata) shows that;

Theorem 4.4.3. *DCCA and Register automata are incomparable in terms of expressive power.*

4.4.2 Many bags

Instead of working with one bag of counters, the automaton can use several bags of counters, much as multiple registers are used in the register automaton. It is easy to formally define CCA with k-bags, using k-tuples of constraints on guards. An interesting fact is that a CCA with k-bags can be converted to a CCA with one bag. This can be achieved because of the following:

- Any CCA, no matter how many bags it has, can be converted to a CCA whose counter values are bounded (We take the maximum constant used in Δ and rewrite the transitions in such a way that we never increment a counter once it reaches that value). This is a direct consequence of Lemma 4.2.10.

- A k-bag CCA whose counters are bounded can be simulated by a CCA with one bag, by using a bit representation. Since the counters are bounded, we know a priori how many bits are needed to represent each bag.

Now we are ready to show that CCA are closed under intersection.

Proposition 4.4.4. *CCA are closed under intersection.*

Proof. Given two CCA A_1 and A_2 with state spaces Q_1 and Q_2 respectively, we construct a CCA A with two bags and state space $Q_1 \times Q_2$ such that A simulates A_1 and A_2. The automaton utilizes its first bag for simulating A_1's counters and second bag for A_2's counters. Now above discussion shows that A can be converted to a CCA with only one bag and hence the proposition. $\square$

4.4.3 Checking any counter

Another strengthening involves checking for the presence of *any* counter satisfying a given constraint and updating it. The idea is to extend the transitions to the following form, $t = (q, a, \tau_0, \tau_1, \ldots, \tau_n, q')$ where each $\tau_i \in C \times \mathsf{Inst} \times \mathbb{N}$ is of the form (c_i, π_i, m_i). The intended semantics of the transition is as follows. Suppose that the current letter is a and data value is d_0. The transition t is enabled if there exist distinct data values $d_1, \ldots, d_n$ such that, for every $i \in [n]_0$, d_i satisfies τ_i. On the occurrence of t each d_i is updated with respect to τ_i. Note that in this way we can modify the counter of a data value which is not the current data value.

Formally a $\mathsf{CCA\ with\ context\ check}$, denoted CCAC, is a tuple (Q, n, Δ, I, F), where the transition relation is modified to be $\Delta \subseteq_{fin} (Q \times \Sigma \times (C \times \mathsf{Inst} \times \mathbb{N})^n \times Q)$ where $n \in \mathbb{N}$.

Let A be a CCAC. A $\mathsf{configuration}$ of A is a pair (q, h), where $q \in Q$ and $h \in \mathcal{B}$. The initial configuration of A is given by (q_0, h_0), where h_0 is the empty bag; that is, $\forall d \in \Gamma, h_0(d) = 0$ and $q_0 \in I$.

Given a data word $w = (a_1, d_1), \ldots (a_m, d_m)$, a run of A on w is a sequence $\gamma = (q_0, h_0)(q_1, h_1) \ldots (q_m, h_m)$ such that $q_0 \in I$ and for all $i, 0 \leq i < m$, there exists a transition $t_i = (q, a, \tau_0, \tau_1, \ldots, \tau_n, q') \in \Delta$ where $\tau_j = (c_j, \pi_j, m_j)$ such that $q = q_i$, $q' = q_{i+1}$, $a = a_{i+1}$ and:

- $h_i(d_{i+1}) \models c_0$ and there exist distinct $e_1, \ldots e_n$ in Γ such that for all $j \in \{1, \ldots, n\}$, $e_j \neq d_{i+1}$ and $h_i(e_j) \models c_j$.
- h_{i+1} is given by:

$$h_{i+1} = \begin{cases} h_i \oplus (d_{i+1}, m') & \text{if } \pi_0 = \mathsf{inc}, m' = h_i(d_{i+1}) + m_0 \\ h_i \oplus (d_{i+1}, m_0) & \text{if } \pi_0 = \mathsf{reset} \\ h_i \oplus (e_j, m') & \text{if } \pi_j = \mathsf{inc}, m' = h_i(e_j) + m_j \\ h_i \oplus (e_j, m_j) & \text{if } \pi_j = \mathsf{reset} \end{cases}$$

We define ω-counter machines with context in a similar way: such a machine is a tuple (Q, Δ, q_0) where Q is finite set of states, q_0 is the initial state and $\Delta \subseteq_{fin} (Q \times (C \times \mathsf{Inst} \times \mathbb{N})^n \times Q)$. A run of an ω-counter machine with context is defined analogously to that of CCA with context. We can then easily show that checking emptiness for CCA with context can be reduced to checking reachability for ω-counter machines with context.

Finally, the following proposition shows that checking emptiness of CCA with context is decidable in EXPSPACE.

Proposition 4.4.5. *Checking non-emptiness of ω-counter machines with context is decidable in EXPSPACE.*

Proof. Given an ω-counter machine $B = (Q, \Delta, q_0)$, we define m_B as in the proof of Proposition 4.3.4.

We construct a Petri net $N_B = (S, T, F, M_0)$ where,

- $S = Q \cup \{i \mid i \in \mathbb{N}, 1 \leq i \leq m_B\}$.
- T is defined according to Δ as follows. Let $t = (q, a, \tau_0, \tau_1, \ldots, \tau_n, q')$ be a transition in Δ where $\tau_j = (c_j, \pi_j, m_j)$ and let $i_0, i_1, \ldots, i_n$ be such that $0 \leq i_j \leq m_B$ and $i_j \models c_j$. Then we add a transition t such that ${}^\bullet t = \{p, i_0, i_1, \ldots, i_n\}$ and $t^\bullet = \{q, i'_0, i'_1, \ldots, i'_n\}$ (take note of the fact that ${}^\bullet t$ and $t^\bullet$ are multisets), where (i) if π_j is inc then $i'_j = min\{m_B, i_j + n_j\}$, and (ii) if π_j is reset then $i'_j = min\{m_B, n_j\}$. Note that i_j can be zero, in which case we add edges only for the places in $[m_B]$.
- The flow relation F is defined according to ${}^\bullet t$ and $t^\bullet$ for each $t \in T$.
- The initial marking is defined as follows. $M_0(q_0) = 1$ and for all p in S, if $p \neq q_0$ then $M_0(p) = 0$.

The rest of the proof is similar to the proof of Proposition 4.3.4 with obvious modifications. $\square$

Given a k-register automaton $A = (Q, \Sigma, \Delta, I, F)$ we can construct a CCA with context which accepts the language $L(A)$.

The way the CCA $A' = (Q', \Sigma, \Delta', q_0', F')$ simulates the register automaton A is as follows. The states of A', namely the set $Q' = Q \times \{0,1\}^k$ stores two kinds of information, the current state of the automaton A and the registers which store a data value (0 indicates the register is holding $\perp$ and 1 indicates the register is holding a data value). When a register write takes place, if the bit corresponding to the written register is 0 it is updated to 1. The information that which data value is in which register is stored in the counter corresponding to the data value. This is done in the following manner. If the counter corresponding to a data value d has value i, $1 \leq i \leq k$, it means that the register i contains the data value d. We also make sure that exactly one counter holds the value i at any time. Suppose Δ contains a read transition (p, a, i, q), we add the set of transitions $\{((p, \bar{v}), a, x = i, [0], (q, \bar{v})) \mid \bar{v} \in \{0,1\}^{i-1} \times \{1\} \times \{0,1\}^{k-i-1}\}$. Suppose Δ contains a write transition (p, a, q, i), we add the set of transitions $(\{p, \bar{v}), a, (x \geq 0), (y = i), (\mathsf{reset}, i), (\mathsf{reset}, 0), (q, \bar{v'}) \mid \bar{v} \in \{0,1\}^k, \bar{v'} = \bar{v} + u_i\}$ to Δ' (u_i is the i-th unit vector). The initial state $q_0' = \big(q_0, 0^k\big)$, and final states are $F' = \{(q, \bar{v}) \mid q \in F, \bar{v} \in \{0,1\}^k\}$. We omit the proof here since it is straightforward. It follows that;

Proposition 4.4.6. *Register automata are strictly weaker than CCA with context in terms of expressiveness.*

4.4.4 The language of constraints

The language of constraints can be strengthened. Previously, the constraints where of the form $x \leq e$ or $x \geq e$. Consider the following language, the language of Presburger arithmetic. The terms in this language are given by the grammar,

$$t ::= 0 \,|\, 1 \,|\, t_1 + t_2 \,|\, x, \;\; x \in V$$

where V is a countably infinite set of variables. The formulas of this language are given by:

$$\varphi ::= t_1 \leq t_2 \,|\, \neg\varphi \,|\, \varphi_1 \vee \varphi_2 \,|\, \exists x.\varphi.$$

The semantics is given as follows. The variables take natural numbers as their values and $+$ is interpreted as addition. We call a formula $\varphi(x)$ with one free variable, a Presburger constraint. We say that $k \in \mathbb{N}$ satisfies $\varphi(x)$ if $k \models \varphi(x)$.

Note that the set of numbers satisfying a constraint may be neither finite nor co-finite. For example, the formula $\exists y.y + y = x$ defines the set of even numbers.

Let C_p be the set of all Presburger constraints. We define CCA with Presburger constraints, abbreviated as CCA + Presburger, as a tuple CCA $= (Q, \Sigma, \Delta, I, F)$, where the transition relation is modified to be $\Delta \subseteq_{fin} (Q \times \Sigma \times C_p \times \mathsf{Inst} \times \mathbb{N} \times Q)$. The definitions of run and acceptance condition is defined in the obvious way.

A set of natural numbers D is *eventually periodic* iff there exists positive numbers m and p such that for all n greater than m, $n \in D$ iff $n + p \in D$. From [End72], we know that the set of numbers satisfying a Presburger constraint is eventually periodic.

Using this, the decision procedure in Section 3 can be modified to check the emptiness of CCA with Presburger constraints. As above, we define ω-counter machines with Presburger constraints: such a machine is a tuple (Q, Δ, q_0) where Q is a finite set of states, $q_0 \in Q$ is the initial state and $\Delta \subseteq_{fin} (Q \times C_p \times \mathsf{Inst} \times \mathbb{N} \times Q)$. Runs are defined in the natural way.

We can then easily show that checking emptiness for CCA with Presburger constraints can be reduced to checking reachability for ω-counter machines with Presburger constraints. Then the following proposition shows that checking emptiness of CCA with Presburger constraints is decidable in EXPSPACE.

Proposition 4.4.7. *Checking non-emptiness of ω-counter machines with Presburger constraints is in* EXPSPACE.

Proof. Given an ω-counter machine $B = (Q, \Delta, q_0)$, let $c_1, \dots c_n$ be the constraints used in Δ. From [End72], we know that $c_1, \dots c_n$ are eventually periodic with the pairs $(m_1, p_1), \dots (m_n, p_n)$. We take $m = m_1 + \dots + m_n$ and p as the least common multiple of $p_1, \dots p_n$.

We construct a Petri net $N_B = (S, T, F, M_0)$ where,

- $S = Q \cup \{i \mid i \in \mathbb{N}, 1 \leq i \leq m + p\}$.
- T is defined according to Δ as follows. Let $(p, c, \pi, n, q) \in \Delta$ and let i be such that $0 \leq i \leq m+p$ and $i \models c$. Then we add a transition t such that ${}^\bullet t = \{p, i\}$ and $t^\bullet = \{q, i'\}$, where (i) if π is inc then $i' = min\{i+n, m+(i+n-m) \bmod p\}$,

and (ii) if π is reset then $i' = min\{n, m + (n - m) \bmod p\}$. Note that i can be zero, in which case we add edges only for the places in $[m_B]$.

- The flow relation F is defined according to ${}^\bullet t$ and $t^\bullet$ for each $t \in T$.
- The initial marking is defined as follows. $M_0(q_0) = 1$ and for all p in S, if $p \neq q_0$ then $M_0(p) = 0$.

The rest of the proof is similar to the proof of Proposition 4.3.4 with obvious modifications. □

4.4.5 Two-way CCA

A two-way CCA is system $(Q, \Sigma, \Delta, I, F)$, where Q, I, F are as usual, the transition relation is $\Delta \subseteq_{fin} (Q \times \Sigma \times C \times \mathsf{Inst} \times \mathbb{N} \times Q \times \{L, R, S\})$. A configuration of A is a triple (q, i, h), where $q \in Q$, $i \in \mathbb{N}$ and $h \in \mathcal{B}$, where the variable i denotes the position of the head. The initial configuration of A is given by $(q_0, 1, h_0)$, where h_0 is the empty bag; that is, $\forall d \in \Gamma, h_0(d) = 0$ and $q_0 \in I$.

Given a data word $w = (a_1, d_1), \ldots (a_n, d_n)$, a run of A on w is a sequence $\gamma = (q_0, i_0, h_0)(q_1, i_1, h_1) \ldots (q_l, i_l, h_l)$ such that $q_0 \in I$ and for all $j, 0 \leq j < l$, there exists a transition $t_j = (q, a, c, \pi, m, q', \mu) \in \Delta$ such that $q = q_j$, $q' = q_{j+1}$, $a = a_{i_j}$ and $h_j(d_{i_j}) \models c$. The resulting counter configuration h_{j+1} is defined as in the case of CCA. Finally, the updated position of the head is determined in the following way;

$$i_{j+1} = \begin{cases} i_j - 1 & \text{if} \quad \mu = L \\ i_j + 1 & \text{if} \quad \mu = R \\ i_j & \text{if} \quad \mu = S \end{cases}$$

We assume that the input word is wrapped with end markers so that if the machine tries to go off the boundary of the word it halts erroneously. We say a run is accepting if the machine halts in a final state.

As we will see below, the emptiness problem is undecidable for the two-way extension of CCAs.

4.4.6 Alternating CCA

An alternating CCA is system $(Q = Q_\forall \uplus Q_\exists, \Delta, I)$, where Q, I, Δ are as usual. Note that there is no designated set of final states; instead, the state set is partitioned into a set of universal states $Q_\forall$ and a set of existential states $Q_\exists$. A configuration of A is a tuple (q, h), where $q \in Q$ and $h \in \mathcal{B}$. The initial configuration of A is given by (q_0, h_0), $q_0 \in I$ and h_0 is the empty bag; that is, $\forall d \in \Gamma, h_0(d) = 0$ and $q_0 \in I$.

Given a data word $w = (a_1, d_1), \ldots (a_n, d_n)$, assume that the automaton is at position i with configuration (q_i, h_i). We say that (q_{i+1}, h_{i+1}) is a valid successor configuration if there exists a transition $t = (q, a, c, \pi, m, q', \mu) \in \Delta$ such that $q = q_i$, $q' = q_{i+1}$, $a = a_{i+1}$ and $h_i(d_{i+1}) \models c$. The resulting counter configuration h_{j+1} is defined as in the case of CCA.

We say that a configuration (q, h) is accepting if

1. $q \in Q_\forall$ and all of its valid successor configurations are accepting. (Note that a configuration with no valid successor configurations is accepting.)

2. $q \in Q_\exists$ and there is a valid successor configuration (q', h') which is accepting.

Finally we say that the word is accepted if the initial configuration (q_0, h_0) is accepting.

Theorem 4.4.8. *The emptiness problem is undecidable for Two-way CCAs and for Alternating CCAs.*

Proof. We do the proofs simultaneously by reducing the Post's Correspondence Problem to the emptiness of two-way CCA and of alternating CCA. Without loss of generality, assume that we are given a PCP instance I which is a set of ordered pairs of non-empty strings over the alphabet $\Sigma = \{l_1, l_2, \ldots l_k\}$, that is $I = \{(u_i, v_i) \mid i \in [n], u_i, v_i \in \Sigma^+\}$. A solution for I is a finite sequence of integers $i_0, i_1, \ldots i_m$, all of which are from the set $\{1, \ldots n\}$ such that $u_{i_0} u_{i_1} \ldots u_{i_m} = v_{i_0} v_{i_1} \ldots v_{i_m}$. We define a two-way CCA which accepts precisely all solutions of I.

For this purpose, we code the PCP solution as a data word, in the following way. Let $\bar{\Sigma} = \{\bar{l_1}, \bar{l_2}, \dots \bar{l_k}\}$ and $\hat{\Sigma} = \Sigma \cup \bar{\Sigma}$. Given a word $w = a_1 a_2 \dots a_n$ in Σ^*, we denote by $\bar{w}$ the word $\bar{a_1}\bar{a_2}\dots\bar{a_n}$ in $\bar{\Sigma}^*$.

A solution of I is a data word w over $\hat{\Sigma}$ such that,

(I) The string projection of the word is in $(u_1\bar{v_1} + u_2\bar{v_2} \dots + u_n\bar{v_n})^+$.

(II) Every data value d occurring in w appears precisely twice, once labelled by a letter from Σ and once by a letter from $\bar{\Sigma}$. Moreover if d is labelled by $l_i \in \Sigma$ in w if and only if it is labelled by $\bar{l_i} \in \bar{\Sigma}$ in v (the second occurrence).

(III) The ordering of data values in the positions labelled by Σ is exactly the same as the ordering of data values in positions labelled by $\bar{\Sigma}$. Formally, let d and e are data values occurring in w. Let d_Σ and e_Σ be the positions where d and e are labelled by letters from Σ. Similarly, let $d_{\bar{\Sigma}}$ and $e_{\bar{\Sigma}}$ be the positions where d and e are labelled by letters from $\bar{\Sigma}$. The condition says that $d_\Sigma < e_\Sigma$ if and only if $d_{\bar{\Sigma}} < e_\Sigma$.

It is easy to see that there is a data word w satisfying the above three conditions iff I has a solution. We show that two-way CCA and alternating CCA can check these three conditions.

1. The first condition is a regular property and can be checked by any finite state automaton. Hence it is easily checked by a CCA.

2. The conjunction of the following four conditions is equivalent to condition (II).

   (a) Data values occurring in Σ-labelled positions are all distinct.

   (b) Data values occurring in $\bar{\Sigma}$-labelled positions are all distinct.

   (c) All data values occurring under $\bar{\Sigma}$-labels occur under Σ-labels as well.

   (d) All data values occurring under Σ-labels occur under $\bar{\Sigma}$-labels as well.

   Note that each of these conditions can be checked by a CCA. Since CCAs are closed under intersection, a CCA can verify condition (II).

3. Condition (III) is checked by a two-way CCA in the following way. We assume that conditions (I) and (II) are verified independently. Given a position i labelled by a letter from Σ we say that the position $j > i$ is the Σ-successor of i iff j is a position labelled by a letter from Σ and all positions k, $i < k < j$ are labelled by letters from $\bar{\Sigma}$. Similarly we can define $\bar{\Sigma}$-successor of a $\bar{\Sigma}$-labelled position. Let i and j be Σ-successors and let d_i and d_j be the corresponding data values. We know that d_i and d_j occur under $\bar{\Sigma}$ as well. Let those positions be $\bar{i}$ and $\bar{j}$. For each Σ-successors i, j the automaton verifies that $\bar{i}$ and $\bar{j}$ are $\bar{\Sigma}$ successors.

   To achieve this, assume that the automaton starts in a Σ position i, it resets the counter of d_i to 1 and goes to next Σ-labelled position j. It increments the counter of d_j to 2. Now, the automaton moves to left end marker and makes a left to right sweep ignoring all Σ positions. During this sweep the automaton stops when it sees the data value d_j under a $\bar{\Sigma}$ label. It resets counter of d_i to zero and then verifies that the next $\bar{\Sigma}$ position has the data value d_j with the help of the counter. After this step the automaton goes to the left end of the word and again makes a right sweep. This time it stops when it sees the data value d_j under a Σ label. Then the procedure is repeated for position j. Finally the machine halts and accepts when it reaches the last Σ position in the data word.

4. Condition (III) is checked by an alternating CCA in the following way. The automaton starts in state q_0. In this state automaton records all the data values it has seen till the current position. Whenever it sees a fresh data value, it makes a universal branching, one branch continues in state q_0 and one branch goes to state q_1. In the state q_1 the automaton verifies the following. Assume the fresh data value d occurs under a Σ label and let the data value on its Σ successor position is e. The automaton verifies that the positions where d and e are occurring under $\bar{\Sigma}$ labels are $\bar{\Sigma}$ successors. This can easily be done by incrementing the counters corresponding to d and e to specially designated values. The q_1 branching halts successfully after each verification. The q_0 branching accepts at the end of the word.

□

In the previous proof, conditions (I), (II) and (III) are in fact verified by a universal CCA. This implies that the emptiness problem for universal CCA is undecidable. Since emptiness problem for universal CCA and universality problem for CCA are equivalent it follows that the universality problem for CCA is undecidable, and hence the language inclusion problem for CCA is undecidable.

4.4.7 Counter acceptance conditions

We compare the expressiveness of CCA and CMA.

Proposition 4.4.9. *The class of CCA-recognizable languages are strictly contained in the class of CMA-recognizable languages.*

Proof. Let $A = (Q, \Sigma, \Delta, I, F)$ be a CCA with m being the maximum constant used in Δ. Let $V = \{0, \dots, m+1\}$. We construct a CMA $A_{cma} = (Q', \Sigma, \Delta', I', F'_l, F'_g)$ where $Q' = Q \times V$, $I' = I \times \{0\}$, $F'_l = Q'$, $F'_g = \{(q, v) \in Q' \mid q \in F\}$. Δ' is defined in the following way,

$$\Delta' = \bigcup_{(q,a,c,\pi,s,q') \in \Delta, (p,v) \in Q'} \left\{ \begin{array}{ll} ((q,w), a, (p,v), (q',v')) & \mid \ v \models c, v' \in V, v' \simeq_{m+1} \pi(v,s) \\ ((q,w), a, \bot, (q',v')) & \mid \ 0 \models c, v' \in V, v' \simeq_{m+1} \pi(0,s) \end{array} \right\}$$

where $\pi(v, s)$ denotes the result of the operation π (one of inc or reset) with argument s on value v and the equivalence is defined as $c \simeq_{m+1} d$ iff $\forall i\, c < m+1 \vee d < m+1 \Rightarrow c = d$. From Lemma 4.2.10 it follows that $L(A) = L(A_{cma})$.

The strict containment follows from the fact that CCA do not accept the language $\overline{L_2}$ (4.2.8) while this language is accepted by a CMA as saw in the last chapter. $\square$

The acceptance condition we have in CCA is *global* in the sense that it relates only to the global control state rather than multiplicities encountered. We can strengthen the acceptance condition as follows: CCA with counter acceptance conditions A is given by $A = (Q, \Sigma, \Delta, I, F, G)$ where Q, Σ, I, Δ, F are as before, and $G \subset_{fin} \mathbb{N}$. We say a final configuration (q, h) is accepting if $q \in F$ and $\forall d \in \Gamma, h(d) \in G$ or $h(d) = 0$.

We then find that the non-emptiness problem continues to be decidable but becomes as hard as Petri net reachability, which is not even known to be elementarily decidable. This is proved by relating this class to that of class memory automata discussed below.

Proposition 4.4.10. *CCA with counter acceptance conditions are expressively equivalent to CMA.*

Proof. The proof of Proposition 4.4.9 can be extended to show that the class of languages recognized by CCA with counter acceptance conditions is contained in the class of CMA-recognizable languages. Let $A = (Q, \Sigma, \Delta, I, F, G)$ be a CCA with counter acceptance condition. Considering A as a CCA construct $A'_{cma} = (Q', \Sigma, \Delta', I', F')$ with m being the maximum constant used in Δ and G as above. Replace the local accepting states $F_l = Q \times G$ to A'_{cma} to get A_{cma}. It is easy to see that $L(A) = L(A_{cma})$.

For the other direction, let $A = (Q, \Sigma, \Delta, I, F_l, F_g)$ be a CMA. Let $Q = \{q_1, q_2, \ldots q_n\}$. We construct a CCA with counter acceptance $A' = (Q', \Sigma, \Delta', I', F, G)$ as follows. We define $Q' = Q$, $I' = I$, $F = F_g$. The accepting counter configurations are defined as $G = \{i \mid q_i \in F_l\}$. The transitions Δ' is given by,

$$\Delta' = \bigcup_{(q_i, a, \tau, q_k) \in \Delta} \left\{ \begin{array}{ll} (q_i, a, x = j, \mathsf{reset}, k, q_k) & \mid \quad \tau = q_j \\ (q_i, a, x = 0, \mathsf{inc}, k, q_k) & \mid \quad \tau = \perp \end{array} \right\}$$

It is easy to see that $L(A) = L(A')$. □

4.5 Discussion

In this chapter we introduced the automaton model CCA. This class of automata is strictly weaker than CMA but at the same time has an elementarily decidable emptiness problem. It is also possible to extend this model to match the expressiveness of CMA.

CCA can accept certain languages, for instance L_a which are not accepted by register automata. The question whether CCA contains register automata is still open. The language $\overline{L_{dd}}$ is accepted by a register automaton, however it is open whether $\overline{L_{dd}}$ is accepted by a CCA. It is possible to extend CCA with context information to include register automata. The language L_{dd} is not accepted by the deterministic subclass of CCA. Since deterministic CCA can accept the language L_a while register automata can not, deterministic CCA and register automata are incomparable in terms of expressiveness.

Regarding the complexity of emptiness checking CCA falls strictly in between register automata and CMA. But with respect to the word problem all these automata have the same complexity.

5

Two-variable logics

5.1 Introduction

In this and subsequent chapters we study two-variable logic for data words. Two-variable logic is the subclass of first-order logic containing formulas which use only two variables x and y. Unlike the full first-order logic whose satisfiability and finite satisfiability problems are undecidable, for two-variable logic both these problems are decidable [Mor75]. More precisely they are complete for NEXPTIME [GKV97]. The expressiveness of this logic is good enough for many applications in AI and natural language processing.

5.2 Preliminaries

In the following, $\mathbb{N}$ denotes the set of natural numbers and $\mathbb{Q}$ the set of rationals. We deal with equivalence relations, preorders and linear orders and briefly introduce them now. Let A be a finite set. An equivalence relation $\sim$ on A is a reflexive, symmetric and transitive relation. A *total preorder* $\leq_p$ on A is a transitive, reflexive, total relation, that is, $u \leq_p v$ and $v \leq_p w$ implies $u \leq_p w$ and for every two elements $u, v \in A$ $u \leq_p v$ or $v \leq_p u$ holds. A *linear order* $\leq_l$ on A is a antisymmetric total preorder, that is, if $u \leq_l v$ and $v \leq_l u$ then $u = v$. Thus, the essential difference between a total preorder and a linear order is that the former allows that for two distinct elements u and v both $u \leq_p v$ and $v \leq_p u$ hold. We call two such elements *equivalent with respect to* $\leq_p$. Thus, a total preorder can be viewed as an equivalence relation $\sim_p$ whose equivalence classes are linearly ordered by $\leq_p$. Clearly, every linear order is a total preorder with equivalence classes of size one. For any element u, the $\sim_p$-class of $u \in A$ is denoted by $[u]_{\sim_p}$ (or $[u]$ if $\sim_p$ is clear from the context). The set of all equivalence classes of $\sim_p$ is denoted by $A/{\sim_p}$.

We only consider finite structures. Therefore, the linear order on the equivalence classes of a total preorder induces a successor relation of the equivalence classes. We write $+1^s_p(u,v)$ if the equivalence class of v with respect to $\leq_p$ is the successor of the equivalence class of u and we call $+1^s_p$ the *induced successor relation* of $\leq_p$. Further we say u and v are $+1_p$-close, if either $u{+}1^s_p v$ or $u \sim_p v$ or $v + 1^s_p u$. If $u \leq_p v$ and if they are not $+1_p$-close, we denote it by $u \ll_p v$. Similarly for $+1_l(u,v)$ and $+1_l$-close.

We use binary relation symbols $\leq_l, \leq_{l_1}, \leq_{l_2}, \dots$ that are always interpreted as linear orders, binary relation symbols $\leq_p, \leq_{p_1}, \leq_{p_2}, \dots$ that are interpreted as total preorders, and binary relation symbols $+1_p, +1_{p_1}, +1_{p_2}, \dots$ as well as $+1_l, +1_{l_1}, +1_{l_2}, \dots$ that are interpreted as successor relations.

A first order structure $\mathfrak{A}$ is a non-empty set A (called the universe) along with some specified binary relations. For example, finite words over the alphabet Σ are (usually) represented as first-order structures of the form $\big([n], (P_a)_{a\in\Sigma}, <, +1\big)$ where $<$ and $+1$ are the order and successor relations on natural numbers (restricted to the set $[n]$) and $(P_a)_{a\in\Sigma}$ are unary predicates representing the Σ labelling on positions. Often while denoting the vocabulary of the structure we abbreviate unary predicates by the alphabet they are representing, for instance $(P_a)_{a\in\Sigma}$ by Σ.

An *ordered structure* is a structure with non-empty universe and some linear orders, some total preorders, some successor relations and some unary relations. We always allow an unlimited number of unary relations and specify the numbers of allowed linear orders and total preorders explicitly. For instance, a $(+1_{l_1}; +1_{p_2}, \leq_{p_2})$-structure is a structure with arbitrarily many unary relations, one successor of linear order and one total preorder together with a corresponding successor relation. We write $(+1_l; +1_p, \leq_p)$ instead of $(+1_{l_1}; +1_{p_2}, \leq_{p_2})$ if no ambiguities arise.

5.2.1 Data words

Given a data word w, the data values define an equivalence relation on the positions of w given by $i \sim j$ if $d_i = d_j$. Thus a data word can be naturally represented as a first-order structure $w = ([n], \Sigma, <, +1, \sim)$.

Assume the data alphabet Γ is linearly ordered by an order relation $<_\Gamma$. In this case data values d_i and d_j on positions i and j can have any of the following relationships: $d_i = d_j$ or $d_i <_\Gamma d_j$ or $d_i >_\Gamma d_j$. This relationship can be expressed by a total preorder on positions given by,

$$i \leq_p j \Leftrightarrow d_i <_\Gamma d_j \text{ or } d_i = d_j.$$

Hence an ordered data word can be represented logically as a first order structure $w = ([n], \Sigma, \leq_l, +1_l, \leq_p, +1_p)$; where $\leq_l$ denotes the linear order on positions and $\leq_p$ denotes the total preorder on positions induced by the order on the data values.

Note that for a linear order and a total preorder the successor relation uniquely defines the order and vice-versa. Therefore even if one of the successor or order relation is absent from the vocabulary, every (ordered) data word has a unique first-order representation in the above mentioned scheme.

Example 5.2.1. *The word ababab is encoded as the structure,*

$$([6], P_a = \{1,3,5\}, P_b = \{2,4,6\}, <, +1)\,.$$

Example 5.2.2. *The data word* $(a, d_2)(b, d_1)(a, d_1)(b, d_2)(a, d_3)(b, d_2)$ *is encoded as the structure,*

$$([6], P_a = \{1,3,5\}, P_b = \{2,4,6\}, <, +1, \sim = \{\{1,4,6\}, \{2,3\}, \{5\}\})\,.$$

Example 5.2.3. *The ordered data word* $(a, 1)(b, 2)(a, 1)(b, 4)(a, 2)(b, 1)$ *is encoded as the structure,*

$$([6], P_a = \{1,3,5\}, P_b = \{2,4,6\}, <, +1, \leq_p)\,,$$

where $\leq_p$ *is the total preorder* $\{1,3,6\} \leq_p \{2,5\} \leq_p \{4\}$.

5.3 Logics

The set of first order (abbreviated as FO) formulas over the vocabulary τ is given by the following syntax;

$$\varphi ::= x = y \mid R(x_1, \ldots, x_n) \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists x\, \varphi$$

where R is an n-ary relation as specified by τ and $x, y, x_1 \ldots$ are first-order variables. The set of monadic second order (abbreviated as MSO) formulas over the vocabulary τ is given by the syntax

$$\varphi ::= x = y \mid R(x_1, \ldots, x_n) \mid X(x) \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists x\, \varphi \mid \exists X\, \varphi$$

where X is a set variable. Note that in MSO variables $X_1, X_2, \ldots$ range over subsets of the universe. Two-variable first-order logic or simply Two-variable logic is the restriction of first order logic to formulas that only use (at most) two variables x and y. We denote two-variable logic by FO^2. Similarly the three-variable logic is denoted by FO^3. Formulas with no free variables are called *sentences*, but in the following we may refer to sentences as formulas when no ambiguity arises.

It is not possible to express in FO^2 that a binary relation R is transitive, a fact easily proved by EF-games. Hence we need to supply the logic with additional non-logical symbols if some relations are to be interpreted as order or equivalence relations. These are specified in the vocabulary. For instance $\mathrm{FO}^2(\Sigma, <, +1)$ is the two variable logic with unary predicates and binary relations $<, +1$ interpreted as a linear order and its successor relation, In other words, this is the two-variable logic on words.

Example 5.3.1. *The following* $\mathrm{FO}^2(\Sigma, <, +1)$ *formula describes that the model (in this case a word) contains three 'a's.*

$$\varphi_1 = \exists x\, (P_a(x) \wedge \exists y\, (x < y \wedge P_a(y) \wedge \exists x\, (y < x \wedge P_a(x))))\,.$$

Example 5.3.2. *The following* $\mathrm{FO}^2(\Sigma, <, +1)$ *formula says that the word is from the language* a^*b^*.

$$\varphi_2 = \forall x \forall y \left(P_a(x) \wedge P_b(y) \rightarrow x < y\right).$$

Example 5.3.3. *The following* $\mathrm{FO}^3(\Sigma, <, +1, \sim)$ *formula over data words describes that between any two positions of the same class there is no 'b'-labelled position from a different class.*

$$\varphi_3 = \forall x \forall y \forall z \left(x \sim y \wedge P_b(z) \wedge x < z \wedge z < y \rightarrow z \sim x\right).$$

Example 5.3.4. *The formula below states that each class contains an 'a' if it contains a 'b' and vice versa.*

$$\varphi_4 = \forall x \left((P_a(x) \rightarrow \exists y \ \ (P_b(y) \wedge x \sim y)) \wedge (P_b(x) \rightarrow \exists y \ \ (P_a(y) \wedge x \sim y))\right)$$

Example 5.3.5. *The following* $\mathrm{FO}^2(\Sigma, <, +1, \leq_p)$ *formula over ordered data words describes that the data values on the positions are non-decreasing.*

$$\varphi_4 = \forall x \forall y \left(x < y \rightarrow x \leq_p y\right).$$

5.3.1 Scott reduction

A very useful property of FO^2 formulas is that they possess a normal form, called Scott Normal Form, with quantifier rank at most two. The following fact is due to Dana Scott [Sco62]. Fix a relational vocabulary τ containing order relations. A formula $\varphi \in \mathrm{FO}^2(\tau)$ is equivalent with respect to satisfiability (as well as finite satisfiability) to a formula of the form;

$$\zeta = \forall x \forall y \ \chi \wedge \bigwedge_{i=1}^{i=k} \forall x \exists y \ \psi_i,$$

where $k \in \mathbb{N}$ and, χ and ψ_i are quantifier free formulas which use only extra unary predicates other than the predicates used in φ. The formula ζ can be obtained from φ in linear time and the size of the formula ζ is linear in terms of the size of φ. Moreover, models of ζ are expansions of models of φ with unary predicates and models of φ are reducts of models of ζ. A full proof of the above statement can be found in [GKV97].

5.4 FO^2 on data words

The primary reason why two-variable logics are looked at in the context of data words is stated below.

Theorem 5.4.1. *Finite satisfiability problem of* $\mathrm{FO}\,(\Sigma, \leq_l, +1_l, \sim_p)$ *is undecidable. More precisely, finite satisfiability problem of* $\mathrm{FO}^3\,(\Sigma, \leq_l, +1_l, \sim_p)$ *is undecidable.*

The above theorem was proved in [BDM+11] which also showed the landmark result that;

Theorem 5.4.2. *Finite satisfiability problem of* $\mathrm{FO}^2\,(\Sigma, \leq_l, +1_l, \sim_p)$ *is decidable and is as hard as reachability of multicounter automata.*

The proof of the above theorem is via automata construction and is interesting in many aspects. Given a formula $\varphi \in \mathrm{FO}^2\,(\Sigma, \leq_l, +1_l, \sim_p)$ it is converted in 2-DEXPTIME to a Data automaton A_φ such that $L\,(\varphi) = L\,(A_\varphi)$. Since checking nonemptiness of Data automaton is decidable it implies that checking (finite) satisfiability of $\mathrm{FO}^2\,(\Sigma, \leq_l, +1_l, \sim_p)$ is decidable. But the complexity of this decision procedure as stated above is as hard as the reachability problem of multicounter automata which is not known to be elementary, making it untenable for practical applications. On the other hand since classical logics provides tools and techniques to test and compare expressiveness questions, this result has great importance.

The proof in [BDM+11] also shows that Data automata are charactersized by the logic $\mathrm{EMSO}^2\,(\Sigma, \leq_l, +1_l, \sim_p, \oplus 1)$ whose formulas are of the form $\exists X_1 \ldots X_n \varphi$ where $\varphi \in \mathrm{FO}^2\,(\Sigma, \leq_l, +1_l, \sim_p, \oplus 1)$ and $X_1, \ldots, X_n$ are set variables. A merit of this proof method is that it allows us to prove decidability without proving a small model property. Note that in this case an elementary small-model property will settle a decades-old problem (is reachability problem for Petri nets elementarily decidable?). In the next two chapters we will emulate this proof method (the history of which dates back to Büchi) to show decidability of other logics.

Next we move on to ordered data words. As mentioned earlier a linear order on data values will imply a total preorder on the positions of the data word. Hence two-variable logic on ordered data words has the signature $\mathrm{FO}^2(\Sigma, \leq_l, +1_l, \leq_p)$. The following was proved in [BDM$^+$11];

Theorem 5.4.3. *Finite satisfiability problem of* $\mathrm{FO}^2(\Sigma, \leq_l, +1_l, \leq_p)$ *is undecidable.*

Even if we replace the preorder $\leq_p$ with its successor relation $+1_p$ the undecidability remains as is shown below.

Theorem 5.4.4. *Finite satisfiability problem of* $\mathrm{FO}^2(\Sigma, \leq_l, +1_l, +1_p)$ *is undecidable.*

Proof. The proof follows the lines of the proof of Proposition 29 in [BDM$^+$11].

We reduce from the Post's Correspondence Problem. Let $I = (u_1, v_1), \ldots, (u_k, v_k)$ be an instance of PCP. We construct an $\mathrm{FO}^2(\leq_l, +1_l; +1_p)$-sentence φ that has a finite model if and only if I has a solution. The sentence φ uses unary predicates from Σ as well as the two unary predicates U, V, and expresses the following conditions:

(1) The string projection of $\leq_l$ is $u_{i_1}v_{i_1} \ldots u_{i_m}v_{i_m}$ for some $m \in \mathbb{N}$. Elements corresponding to some u_i and v_i are marked with U and V, respectively.

(2) Every equivalence class of $+1_p$ contains exactly two elements such that

   - One is marked with U and one is marked with V.
   - Both carry the same label from Σ.

(3) Positions $x_1, \ldots, x_{|u|}$ corresponding to the positions of $u := u_{i_1} \ldots u_{i_m}$ fulfill $+1_p(i, i+1)$ for all $i \in \{1, \ldots, |u| - 1\}$. Analogously for v.

Condition (1) can be expressed in the following way. Given a string u_iv_i, it is straightforward to write a formula $\varphi_{u_iv_i}(x) \in \mathrm{FO}^2(\Sigma, +1_l)$ which states that there is a subword u_iv_i starting from the position x where positions of u_i are labelled by U and positions of v_i are labelled by V. In addition, the subword is followed by a U position unless the word ends. Next we state that;

$$\forall x \left(U(x) \wedge (\exists y \ \ (+1_l(y,x) \wedge V(y)) \vee \neg \exists y \ \ +1_l(y,x)) \rightarrow \bigvee_{i \in k} \varphi_{u_i v_i}(x) \right)$$

The second condition is ensured by the formulas;

$$\neg \exists x \exists y \, (x \sim_p y \wedge x \neq y \wedge ((U(x) \wedge U(y)) \vee (V(x) \wedge V(y))))$$
$$\forall x \bigwedge_{a \in \Sigma} (P_a(x) \wedge U(x) \rightarrow \exists y \ \ (P_a(y) \wedge x \sim_p y \wedge V(y)))$$
$$\forall x \bigwedge_{a \in \Sigma} (P_a(x) \wedge V(x) \rightarrow \exists y \ \ (P_a(y) \wedge x \sim_p y \wedge U(y)))$$

The third condition can be ensured by the formula,

$$\forall x \forall y (U(x) \wedge U(y) \wedge +1_p(x,y) \rightarrow x <_l y)$$

Now, from a solution $\vec{i} = i_1 \ldots i_m$ a model of φ can be constructed easily. On the other hand, let $\mathcal{M}$ be a a model of φ. By (1), the string projection of $\mathcal{M}$ is of the form $u_{i_1} v_{i_1} \ldots u_{i_m} v_{i_m}$. The U- and V-labeled elements are ordered with respect to $\leq_p$ due to (3). Thus, (2) implies that $u_{i_1} \ldots u_{i_m} = v_{i_1} \ldots v_{i_m}$. □

This means that for two-variable logic to be decidable on ordered data words either the linear order $\leq_l$ or the successor relation $+1_l$ has to be dropped from the vocabulary. Following this line in [SZ10, SZ11] it was shown that,

Theorem 5.4.5. *Finite satisfiability problem of* $\mathrm{FO}^2\,(\Sigma, \leq_l, \leq_p, +1_p)$ *is decidable in* EXPSPACE.

The above theorem is proved by showing a small model property. In the subsequent chapters we consider the other line that is to drop $\leq_l$. The status of finite satisfiability problem for $\mathrm{FO}^2\,(\Sigma, +1_l, \leq_p, +1_p)$ is still open. In the next chapter we restrict the preorder to be a linear order and study the logic with two linear orders, namely $\mathrm{FO}^2\,(\Sigma, \leq_{l_1}, +1_{l_1}, +1_{l_2})$ and its subclasses. While this is the two-variable logic on class of ordered data words where all data values appearing in the word are different, this logic is interesting in its own way as described in the next paragraph.

The status of satisfiability problem of first-order logic on ordered structures is very interesting as these are one of the simplest mathematical structures and at the same time ubiquitous in computer science as they naturally arise in computation. To give a short account of the results in this direction, in [EVW02] it is shown that the satisfiability and finite satisfiability problems of FO^2 over words are NEXPTIME-complete. In [Ott01] the following are shown. The logic FO^2 over ordered or well-ordered domains, or in the presence of one well-founded relation, is decidable for satisfiability as well as for finite satisfiability. The complexity of these decision problems is essentially the same as for plain unconstrained FO^2. In contrast, FO^2 becomes undecidable for satisfiability and for finite-satisfiability, if a sufficiently large number of predicates (at least eight) are required to be interpreted as orderings, well-orderings, or as arbitrary well-founded relations. In [KO05] it is shown that FO^2 with two transitive relations (without equality) is undecidable. In [KO05] it is shown that FO^2 is undecidable with three equivalence relations, but is decidable when the number of equivalence relations is two. Later in [KT09] it is shown that in the case of two equivalence relations, finite satisfiability is decidable in 3-EXPTIME. In the same paper the undecidability is sharpened to one equivalence relation and one transitive relation.

As a warm-up, we show the following theorem. Note that $+2_{l_1}$ denotes the second-successor or successor-of-successor relation in the linear order $\leq_{l_1}$. Similarly for $+3_{l_1}$.

Theorem 5.4.6. *The finite satisfiability problems for the following logics are undecidable.*

(a) $\mathrm{FO}^2\left(\Sigma, \leq_{l_1}, +1_{l_1}, \leq_{l_2}, +1_{l_2}\right)$

(b) $\mathrm{FO}^3\left(\Sigma, +1_{l_1}, +1_{l_2}\right)$

(c) $\mathrm{FO}^2\left(\Sigma, +1_{l_1}, +2_{l_1}, +3_{l_1}, +1_{l_2}, +2_{l_2}\right)$

Proof. We reduce the Post's Correspondence Problem (PCP) to the finite satisfiability problems of the logics $\mathrm{FO}^2\left(\Sigma, +1_{l_1}, \leq_{l_1}, +1_{l_2}, \leq_{l_2}\right)$, $\mathrm{FO}^3\left(\Sigma, +1_{l_1}, +1_{l_2}\right)$ and $\mathrm{FO}^2\left(\Sigma, +1_{l_1}, +2_{l_1}, +3_{l_1}, +1_{l_2}, +2_{l_2}\right)$. The variant of PCP in which the strings are of length one or two is also undecidable [HU79]. We employ this variant for the reduction. Assume that we are given a PCP instance $I = \{(u_i, v_i) \mid i \in [n], u_i, v_i \in \Sigma^{\leq 2}\}$

over the alphabet $\Sigma = \{l_1, l_2, \ldots l_k\}$. We encode the PCP solution as structures in the above vocabularies, in the following way. Let $\Sigma' = \{l'_1, l'_2, \ldots l'_k\}$ and $\hat{\Sigma} = \Sigma \cup \Sigma'$. Given a word $w = a_1 a_2 \ldots a_n$ in Σ^*, we denote by w' the word $a'_1 a'_2 \ldots a'_n$ in Σ'^*.

A solution of I is a structure $\mathcal{A} = (A, \hat{\Sigma}, +1_{l_1}, +1_{l_2})$ over $\hat{\Sigma}$ such that,

(1) The word $(A, \hat{\Sigma}, +1_{l_1})$ is in the language $(u_1 v'_1 + u_2 v'_2 \ldots + u_n v'_n)^+$. This language is expressible in FO^2 $(\hat{\Sigma}, +1_{l_1})$ as in the proof of Theorem 5.4.4, let us call it φ_1.

(2) The word $(A, \hat{\Sigma}, +1_{l_2})$ is in the language $(l_1 l'_1 + l_2 l'_2 \ldots + l_k l'_k)^+$. This language is expressible in FO^2 $(\hat{\Sigma}, +1_{l_2})$ by the formulas (call them φ_2),

$$\forall x \forall y \left(\bigwedge_i \left(P_{l_i}(x) \wedge +1_{l_2}(x, y) \rightarrow P_{l'_i}(y) \right) \wedge \bigwedge_i \left(P_{l'_i}(x) \wedge +1_{l_2}(x, y) \rightarrow P_{l_i}(y) \right) \right)$$

$$\exists x \left(\neg \left(\exists y \; +1_{l_2}(y, x) \right) \rightarrow \bigvee_i P_{l_i}(x) \right) \wedge \exists x \left(\neg \left(\exists y \; +1_{l_2}(x, y) \right) \rightarrow \bigvee_i P_{l'_i}(x) \right)$$

(3a) The third condition is specific for each of the logics, though they all express the same form of matching between Σ and Σ' positions. We say x is Σ-position, denoted as $\Sigma(x)$, if it is labeled by a letter from Σ, that is if $P_{l_1}(x) \vee P_{l_2}(x) \ldots \vee P_{l_k}(x)$ is true. Similarly, we say x is a Σ'-position, denoted as $\Sigma'(x)$, if $P_{l'_1}(x) \vee P_{l'_2}(x) \ldots \vee P_{l'_k}(x)$ is true. Our next condition says that, taken only the Σ positions, the order $\leq_{l_2}$ respects the order $\leq_{l_1}$, similarly is the case with Σ' positions. This can be expressed by the following formula in FO$^2 \left(\hat{\Sigma}, +1_{l_1}, \leq_{l_1}, +1_{l_2}, \leq_{l_2} \right)$,

$$\varphi_{3a} \equiv \forall xy \;\; ((\Sigma(x) \wedge \Sigma(y) \wedge x \leq_{l_1} y \rightarrow x \leq_{l_2} y)$$

$$\wedge \; (\Sigma'(x) \wedge \Sigma'(y) \wedge x \leq_{l_1} y \rightarrow x \leq_{l_2} y))$$

(3b) Let $S(x, y)$ be true if either one of the following conditions holds : (1) both x and y are Σ positions and no position between x and y in $+1_{l_1}$ is labeled from Σ. (2) Analogously, both x and y are Σ' positions and no position between x and y in $+1_{l_1}$ is labeled from Σ'. Notice that $S(x, y)$ can be coded in $FO^3(\hat{\Sigma}, +1_{l_1}, +1_{l_2})$ since the distance between any two consecutive Σ positions or any two consecutive Σ' positions is bounded by two. The formula $S(x, y) = S_\Sigma(x, y) \vee S_{\Sigma'}(x, y)$. Below we give the definition of $S_\Sigma(x, y)$ while $S_{\Sigma'}(x, y)$ is defined analogously.

$$\begin{aligned} S_\Sigma(x,y) = \; & (\Sigma(x) \wedge \Sigma(y)) \wedge \\ & (+1_{l_1}(x,y) \\ & \quad \vee \exists z\, (+1_{l_1}(x,z) \wedge \Sigma'(z) \wedge +1_{l_1}(z,y)) \\ & \quad \vee \exists z\, (+1_{l_1}(x,z) \wedge \Sigma'(z) \wedge \exists x\, (+1_{l_1}(z,x) \wedge \Sigma'(x) \wedge +1_{l_1}(x,y)))) \end{aligned}$$

Once we have S we enforce the correct matching in the following way, φ_{3b} is the conjunction of the following formulas in $FO^3(\hat{\Sigma}, +1_{l_1}, +1_{l_2})$,

$$\forall xyz((\Sigma(x) \wedge \Sigma(y) \wedge \Sigma'(z) \wedge S(x,y) \wedge x + 1_{l_2} z) \rightarrow z + 1_{l_2} y)$$

$$\forall xyz((\Sigma'(x) \wedge \Sigma'(y) \wedge \Sigma(z) \wedge S(x,y) \wedge x + 1_{l_2} z) \rightarrow z + 1_{l_2} y)$$

(3c) Note that, when the strings are of length at most two, the predicate $S(x, y)$ defined above, can be coded by using the successor relations $+1_{l_1}$, $+2_{l_1}$ and $+3_{l_1}$ as in the previous case. Again, we define $S(x, y) = S_\Sigma(x, y) \vee S_{\Sigma'}(x, y)$ and $S_\Sigma(x, y)$ is;

$$\begin{aligned} S_\Sigma(x,y) = \; & (\Sigma(x) \wedge \Sigma(y)) \wedge \\ & (+1_{l_1}(x,y) \\ & \quad \vee (+2_{l_1}(x,y) \wedge \exists y\, (+1_{l_1}(x,y) \wedge \Sigma'(y))) \\ & \quad \vee (+3_{l_1}(x,y) \wedge \exists y\, (+2_{l_1}(x,y) \wedge \Sigma'(y)) \wedge \exists y\, (+1_{l_1}(x,y) \wedge \Sigma'(y)))) \end{aligned}$$

The matching is done by φ_{3c} which is a conjunction of the following formulas in $FO^2\left(\hat{\Sigma}, +1_{l_1}, +2_{l_1}, +3_{l_1}, +1_{l_2}, +2_{l_2}\right)$,

$$\forall xy \;\; ((\Sigma(x) \wedge \Sigma(y) \wedge S(x,y)) \rightarrow x + 2_{l_2} y)$$

$$\forall xy \;\; ((\Sigma'(x) \wedge \Sigma'(y) \wedge S(x,y)) \rightarrow x + 2_{l_2} y)$$

We claim that the formulas $\varphi_1 \wedge \varphi_2 \wedge \varphi_{3a}, \varphi_1 \wedge \varphi_2 \wedge \varphi_{3b}, \varphi_1 \wedge \varphi_2 \wedge \varphi_{3c}$ encodes a solution of I in the logics $\mathrm{FO}^2\left(\hat{\Sigma}, +1_{l_1}, \leq_{l_1}, +1_{l_2}, \leq_{l_2}\right)$, $\mathrm{FO}^3\left(\hat{\Sigma}, +1_{l_1}, +1_{l_2}\right)$, $\mathrm{FO}^2\left(\hat{\Sigma}, +1_{l_1}, +2_{l_1}, +3_{l_1}, +1_{l_2}, +2_{l_2}\right)$ respectively. That is I has a solution if and only if each of them is satisfiable. Suppose I has a solution $i_0, i_1, \dots i_m$, in which case $u_{i_0}u_{i_1}\dots u_{i_m} = v_{i_0}v_{i_1}\dots v_{i_m}$, call it w. Let $|w| = n$. We define the structure $\left([2n], \hat{\Sigma}, +1, +1_{l_2}\right)$ such that $+1$ is the successor relation on $[2n]$ and $\left([2n], \hat{\Sigma}, +1\right)$ is the word $u_{i_0}v'_{i_0}\dots u_{i_m}v'_{i_m}$. Note that in this word there are n-many Σ positions and Σ' positions. Let those be the sequences $\sigma_1 \dots \sigma_n$ and $\sigma'_1 \dots \sigma'_n$ in the ascending order. Define the order $+1_{l_2}$ as $\sigma_1\sigma'_1\sigma_2\sigma'_2\dots\sigma_n\sigma'_n$. Clearly the structure satisfies all the three conditions. Now suppose a structure satisfies all the three conditions. Without loss of generality we can assume that it is of the form $\left([2n], \hat{\Sigma}, +1, +1_{l_2}\right)$ for some $n \in \mathbb{N}$ such that $\left([2n], \hat{\Sigma}, +1\right)$ is a word of the form $u_{i_0}v'_{i_0}\dots u_{i_m}v'_{i_m}$ for some $i_0 \dots i_m$. Let $\sigma_1 \dots \sigma_n$ and $\sigma'_1 \dots \sigma'_n$ be the Σ and Σ' positions in the ascending order. Condition (3) ensures that for every i, $\sigma_i + 1_{l_2}\sigma'_i + 1_{l_2}\sigma_{i+1}$ (if σ_{i+1} exists) and condition (2) ensures that σ_i is labelled by letter 'l' if and only if σ_i is labelled by 'l''. Together it implies that $u_{i_0}u_{i_1}\dots u_{i_m} = v_{i_0}v_{i_1}\dots v_{i_m}$. □

Note that undecidability of $\mathrm{FO}^3(\Sigma, +1_{l_1}, +1_{l_2})$ also implies undecidability of $\mathrm{FO}^3(\Sigma, \leq_{l_1}, \leq_{l_2})$ since in three variables the successor relation $+1_{l_1}$ is expressible in terms of the order relation $\leq_{l_1}$. An interesting question is to sharpen the undecidability of $\mathrm{FO}^2(\Sigma, +1_{l_1}, +2_{l_1}, +3_{l_1}, +1_{l_2}, +2_{l_2})$ by reducing the number of successors required. In the next chapter we will show that $\mathrm{FO}^2(\Sigma, +1_{l_1}, +1_{l_2})$ is decidable.

6

Two-successor structures

6.1 Introduction

In this chapter we study the finite satisfiability problem of two variable logic on first order structures with two or more successor relations. Our approach is automata theoretic. After necessary definitions, we define an automaton formalism on structures with two successor relations. An algorithm for deciding the non-emptiness of the language of the automaton is proved. The decidability of the satisfiability of the logic follows from an equivalence between the logic and the automata in terms of the language defined. Next, we move on to structures with more than two successors and generalize the automata whose decidability of non-emptiness remains open.

6.2 Preliminaries

As usual, we denote by $[n]$ the set $\{1, \ldots, n\}$ and whenever $+1_l$ is associated with this set we mean the usual successor relation on $[n]$.

A two-successor structure (abbreviated as 2-SS) $\mathfrak{A}$ over Σ is a first order structure $\mathfrak{A} = (A, \lambda, +1_{l_1}, +1_{l_2})$ where A is a finite set, $\lambda : A \to \Sigma$ is a labeling function, $+1_{l_1}, +1_{l_2}$ are successor relations of two linear orders over A. We denote the linear order corresponding to $+1_{l_1}$ (alternatively $+1_{l_2}$) by the symbol $\leq_{l_1}$ (alternatively

$\leq_{l_2}$). Restricting the structure $\mathfrak{A}$ to either of the orders yields a word, we call the word $(A, \lambda, +1_{l_1})$ the projection of $\mathfrak{A}$ to the order $+1_{l_1}$.

Given any 2-SS $\mathfrak{A} = (A, \lambda, +1_{l_1}, +1_{l_2})$ where $|A| = n$ we can rewrite $\mathfrak{A}$ uniquely as $\big([n], \lambda', +1_l, +1'_{l_2}\big)$ such that $\lambda' = \kappa^{-1} \circ \lambda$ and $+1'_{l_2} = \{(\kappa(x), \kappa(y)) \mid x + 1_{l_2} y\}$ where κ is the unique isomorphism from $(A, +1_{l_1})$ to $([n], +1_l)$. Similarly, it can be also rewritten uniquely as $\big([n], \lambda'', +1'_{l_1}, +1_l\big)$.

6.3 Automata on 2-SS

Given a 2-SS of the form $\mathfrak{A} = ([n], \lambda, +1_{l_1} = +1_l, +1_{l_2})$, let $([n], \lambda, +1_l) = a_1 a_2 \ldots a_n$ be the projection of $\mathfrak{A}$ to the order $+1_{l_1}$. We define the marked string projection of $\mathfrak{A}$ to $+1_{l_1}$, abbreviated as $msp_{+1_{l_1}}(\mathfrak{A})$, as the word $(a_1, b_1)(a_2, b_2) \ldots (a_n, b_n)$ where $b_i \in \{-1, 0, 1\}$, such that

$$b_i = \begin{cases} -1 & \text{if } 1 \leq i < n \text{ and } +1_{l_2}((i+1), i), \\ \ \ 1 & \text{if } 1 \leq i < n \text{ and } +1_{l_2}(i, (i+1)), \\ \ \ 0 & \text{otherwise.} \end{cases}$$

Given any 2-SS $\mathfrak{A}$ we can define its $msp_{+1_{l_1}}(\mathfrak{A})$ by converting it into the above form.

Similarly, we can define the marked string projection of $\mathfrak{A}$ to $+1_{l_2}$ denoted as $msp_{+1_{l_2}}(\mathfrak{A})$. For this, we first convert it into the form $\mathfrak{A}' = ([n], \lambda, +1_{l_1}, +1_{l_2} = +1_l)$. Let $([n], \lambda, +1_l) = a_1 a_2 \ldots a_n$ be the projection of $\mathfrak{A}'$ to the order $+1_{l_2}$. $msp_{+1_{l_2}}(\mathfrak{A}) = msp_{+1_{l_2}}(\mathfrak{A}')$ is defined as the word $(a_1, b_1)(a_2, b_2) \ldots (a_n, b_n)$ where $b_i \in \{-1, 0, 1\}$, such that

$$b_i = \begin{cases} -1 & \text{if } 1 \leq i < n \text{ and } +1_{l_1}((i+1), i), \\ \ \ 1 & \text{if } 1 \leq i < n \text{ and } +1_{l_1}(i, (i+1)), \\ \ \ 0 & \text{otherwise.} \end{cases}$$

In the following we define the notion of a 2-SS automaton. Fix an alphabet Σ. A 2-SS automaton $A = (B, C)$ is a composite automaton consisting of two word automata B and C. The automaton B is a non-deterministic letter-to-letter word transducer with the input alphabet $\Sigma \times \{-1, 0, 1\}$ and an output alphabet Σ' (included in the definition of B). The automaton C is a non-deterministic finite state recognizer accepting words over the alphabet Σ'. Given a 2-SS $\mathfrak{A} = (A, \lambda, +1_{l_1}, +1_{l_2})$ the automaton works as follows. The transducer B runs over the $msp_{+1_{l_1}}(\mathfrak{A})$ yielding a string $w = (A, \lambda', +1_{l_1})$ in Σ'^*, where $\lambda' : A \to \Sigma'$. The automaton C runs over the string $w' = (A, \lambda', +1_{l_2})$, notice that w is permuted to the order $+1_{l_2}$. Finally, the automaton A accepts $\mathfrak{A}$ if both B and C have a successful run, that is they both finish in one of their final states respectively.

Definition 6.3.1. *Formally, a 2-SS automaton A is a tuple $A = (B, C)$, where B is a word transducer given by the tuple $B = (Q_b, (\Sigma \times \{-1, 0, 1\}), \Sigma', O_b, \Delta_b, I_b, F_b)$, where Q_b is the finite set of states, $(\Sigma \times \{-1, 0, 1\})$ is the input alphabet, Σ' is the output alphabet, $I_b \subseteq Q_b$ is the set of initial states, $F_b \subseteq Q_b$ is the set of final states, $\Delta_b \subseteq Q_b \times (\Sigma \times \{-1, 0, 1\}) \times Q_b$ is the set of transitions and $O_b : \Delta_b \to \Sigma'$ is the output function.*

The automaton C is given by the tuple $C = (Q_c, \Sigma', \Delta_c, I_c, F_c)$ where Q_c is the finite set of states, Σ' is the alphabet, $I_c \subseteq Q_c$ is the set of initial states, $F_c \subseteq Q_c$ is the set of final states, $\Delta_c \subseteq Q_c \times \Sigma' \times Q_c$ is the set of transitions.

Given a marked string $w = (a_1, b_1)(a_2, b_2) \ldots (a_n, b_n)$ we define a run ρ_B of B as a sequence $q_0 q_1 \ldots q_n$ such that $q_0 \in I_b$ and for every $i \in [n]$ there is a transition $\delta_i = (p, (a, b), q)$ in Δ_b such that $q_{i-1} = p$, $q_i = q$, $a_i = a$ and $b_i = b$. The run ρ_b is accepting if $q_n \in F_b$. Given any accepting run ρ_b of B on w, it uniquely defines an output string $w' = a'_1 a'_2 \ldots a'_n \in \Sigma'^*$ where $a'_i = O_b(\delta_i)$. Given a word $w' = a'_1 a'_2 \ldots a'_n \in \Sigma'^*$ a run ρ_c of C is a sequence $q_0 q_1 \ldots q_n$ such that $q_0 \in I_c$ and for every $i \in [n]$ there is a transition (p, a', q) in Δ_c such that $q_{i-1} = p$, $q_i = q$ and $a'_i = a'$. The run ρ_c is accepting if $q_n \in F_c$. Now, we define the run ρ of A on the 2-SS $\mathfrak{A} = (A, \lambda, +1_{l_1}, +1_{l_2})$ as a pair (ρ_b, ρ_c) such that (i) ρ_b is an accepting run of B on $msp_{+1_{l_1}}(\mathfrak{A})$ yielding a word $(A, \lambda', +1_{l_1})$ and (ii) ρ_c is an accepting run of C on the word $(A, \lambda', +1_{l_2})$.

We look at some example languages.

Example 6.3.2. *Let L_1 be the language of 2-SS's where both the orders coincide, that is $L_1 = \{\mathfrak{A} \mid \mathfrak{A} \models \forall xy \ (x \leq_{l_1} y \Leftrightarrow x \leq_{l_2} y)\}$. Observe that in the string projection $msp_{+1_{l_1}}$ all positions except the last position is labelled by the marking*

1. *The last position is marked by* 0*. The automaton* B *verifies the marking and accepts the structure. The automaton* C *always accepts.*

What is more interesting is that we can accept 2-SS's whose string projections are non-regular.

Example 6.3.3. *Consider the set of* 2*-SS's such that,*

- *the word projection of the* 2*-SS to the order* $+1_{l_1}$ *belongs to the language* $a^*b^*c^*$*,*
- *the projection to* $+1_{l_2}$ *belongs to the language* $(abc)^*$*.*

The above conditions are checked easily by the automata B *and* C*. Notice that the projection of the* 2*-SS to* $+1_{l_1}$ *has to be the language* $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ *which is not regular.*

Lemma 6.3.4. *Given a regular language* $L \subseteq \Sigma^*$*, there is a* 2*-SS automaton accepting all* 2*-SS's whose projections to* $+1_{l_1}$ *is in* L*. Similarly, there is a* 2*-SS automaton accepting all* 2*-SS's whose projections to* $+1_{l_2}$ *is in* L*.*

Proof. In the first case the transducer B checks if the projection of the 2-SS to $+1_{l_1}$ (ignoring the markings) is in L and C accepts Σ'^*. For the second case, the transducer B simply copies the string (again ignoring the markings) and C accepts if its input is in L. □

Lemma 6.3.5. *Languages recognized by* 2*-SS automata are closed under union, intersection and renaming.*

Proof. We deal with closure under intersection first. Assume that we are given two 2-SS automata $A_1 = (B_1, C_1)$ and $A_2 = (B_2, C_2)$ with internal alphabets Σ_1 and Σ_2 respectively. Without loss of generality assume that states of the constituent automata and the internal alphabets are (pair-wise) disjoint. If the sets of states or the alphabets are not disjoint we simply rename them appropriately.

For intersection, we define the internal alphabet of the product automata as $\Sigma' = \Sigma_1 \times \Sigma_2$. We define the intersection of A_1 and A_2 as $A_\cap = (B, C)$ where B

is the product of B_1 and B_2 and C is the product of C_1 and C_2 which are defined as follows. The set of states of B is the product of states of B_1 and B_2 and the set of initial (alt. final) states of B is the product of set of initial (alt. final) states of B_1 and B_2. The transition $\delta = ((p_1, p_2), a, (q_1, q_2))$ belongs to the set of transitions of B if $\delta_1 = (p_1, a, q_1)$ and $\delta_2 = (p_2, a, q_2)$ are in the sets of transitions of B_1 and B_2 respectively. Finally the automaton outputs $(a_1, a_2) \in \Sigma'$ if B_1 outputs a_1 on δ_1 and B_2 outputs a_2 on δ_2. For the automaton C, we take the set of states as the product of sets of states of C_1 and C_2, the set of initial and final states as the product of sets of initial and final states of C_1 and C_2 respectively. The automaton C has a transition $\delta = ((p_1, p_2), (a_1, a_2), (q_1, q_2))$ if C_1 has a transition $\delta_1 = (p_1, a_1, q_1)$ and C_2 has a transition $\delta_2 = (p_2, a_2, q_2)$.

For union of A_1 and A_2 the construction is similar. The internal alphabet is defined to be $\Sigma' = \Sigma_1 \cup \Sigma_2$. Define the union of A_1 and A_2 as $A_\cup = (B, C)$ where B is the union of B_1 and B_2 and C is the union of C_1 and C_2 which are defined as follows. The set of states of B is the union of states of B_1 and B_2 and the set of initial (alt. final) states of B is the union of set of initial (alt. final) states of B_1 and B_2. The set of transitions of B is the union of sets of transitions of B_1 and B_2. Similarly the output function is the union of output functions of B_1 and B_2. For the automaton C, we take the union of automata C_1 and C_2.

For showing closure under renaming, let $A = (B, C)$ be a 2-SS automaton over Σ and let $h : \Sigma_1 \to \Sigma$ be a letter-to-letter renaming. The language $h^{-1}(L(A))$ is obtained by the automaton $A' = (B', C)$ where B' is B with the following changes. We assign the alphabet of B' as Σ_1 and leave the set of states as well as the sets of initial and final states unchanged. For each transition $\delta = (p, a, q)$ of B, we add the set of transitions $\{(p, a_1, q) \mid a_1 \in h^{-1}(a)\}$ to B'. On these transitions the automaton outputs $O(\delta)$ where O is the output function of B.

□

Example 6.3.6. *Consider the language L_M, the collection of 2-SS's such that,*

- *the projection to $+1_{l_1}$ is in $\$_1 a^+ \#_1 \$_2 b^+ \#_2$,*
- *projection to $+1_{l_2}$ is in $\$_1 \$_2 (ab)^+ \#_1 \#_2$,*

- *there exist two positions x_0, x_1 having the same label from $\{a, b\}$ such that $x_0 \leq_{l_1} x_1$ and $x_1 \leq_{l_2} x_0$.*

The language, L_M is accepted by a 2-SS automaton. Conditions 1 and 2 can be checked easily by B and C. For condition 3, the transducer B non-deterministically chooses two positions having the same label (either a or b), $x_0 \leq_{l_1} x_1$ and outputs 0 at x_0 and 1 at x_1 and \$ at every other position. The automaton C verifies that its input is of the form $\$^*1\$^*0\$^*$.

Proposition 6.3.7. *The complement of the language L_M (denoted as $\overline{L_M}$) is not accepted by any 2-SS automaton.*

Proof. For the sake of contradiction, assume that there is a 2-SS automaton $A = (B, C)$ accepting the language $\overline{L_M}$. Let the number of states in B be n. Consider the 2-SS $\mathfrak{A} = ([2k+4], \lambda, +1_l, +1_{l_2})$ such that $([2k+4], \lambda, +1_l)$ is the word $\$_1 a^k \#_1 \$_2 b^k \#_2$ and $+1_{l_2}$ is the successor relation;

$$\{(1, k+3), (k+3, 2), (2, k+4), (k+4, 3) \dots (k+2, 2k+4)\}$$

where $k > n$. This 2-SS is shown in the Figure 6.1, the relation $+1_l$ is shown in black and $+1_{l_2}$ is shown in blue. Note that in the $\mathrm{msp}_{+1_{l_1}}(\mathfrak{A})$ all the markings are zero. Since $\mathfrak{A}$ is in $\overline{L_M}$, there is an accepting run of B such that there exist two positions $i < j$ with label a and $q_{i-1} = q_{j-1}$ in the run. We define the order $+1'_{l_2}$ as,

$$\begin{aligned} &\{(l, k+2+l) \mid 1 \leq l \leq k+2, l \neq i, l \neq j\} \\ &\cup \{(k+2+l, l+1) \mid 1 \leq l < k+2, l+1 \neq i, l+1 \neq j\} \\ &\cup \{(k+1+i, j), (j, k+2+i), (k+1+j, i), (i, k+2+j)\} \end{aligned}$$

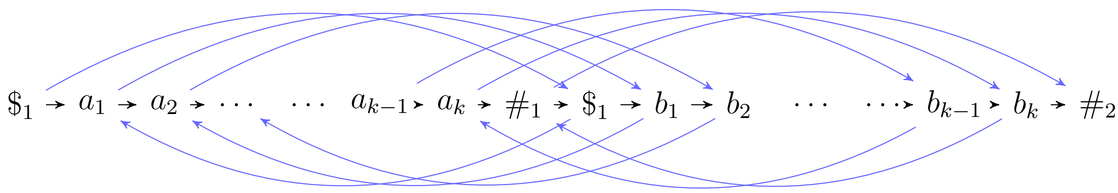

Figure 6.1: The initial 2-SS in Proposition 6.3.7

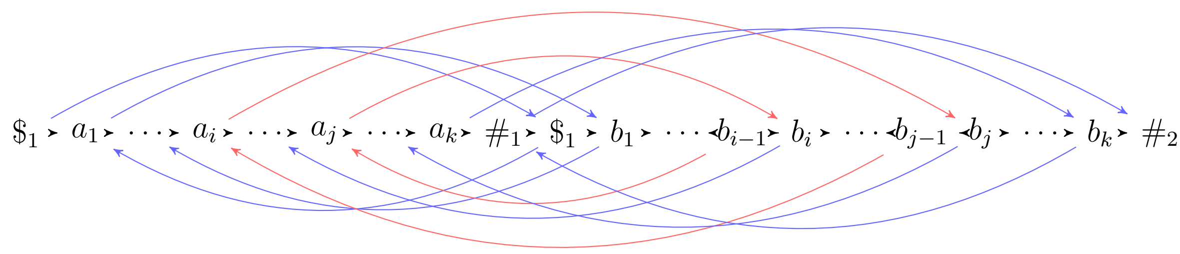

Figure 6.2: The modified 2-SS in Proposition 6.3.7

In the relation $+1'_{l_2}$ only the positions i and j are switched from $+1_{l_2}$. Let $\mathfrak{A}' = \left([2k+4], \lambda, +1_l, +1'_{l_2}\right)$ (shown in Figure 6.2, the switched edges are shown in red). It is the case that $\mathrm{msp}_{+1_{l_1}}(\mathfrak{A}) = \mathrm{msp}_{+1_{l_1}}(\mathfrak{A}')$ and B has an accepting run on $\mathrm{msp}_{+1_{l_1}}(\mathfrak{A}')$ outputting the same string as in the case of $\mathfrak{A}$, which then permuted to $+1_{l_2}$ and $+1'_{l_2}$ gives the same string. Hence C also has an accepting run. But, $\mathfrak{A}'$ does not belong to $\overline{L_M}$, leading to a contradiction. $\square$

This shows that,

Lemma 6.3.8. *The class of languages accepted by 2-SS, automata are not closed under complementation.*

Using a similar argument, we can show that the class of 2-SS automata where the transducer B is deterministic is strictly weaker.

6.3.1 Reducing 2-SS automata to $\mathrm{EMSO}^2\left(\Sigma, +1_{l_1}, +1_{l_2}\right)$

Proposition 6.3.9. *Given a 2-SS automaton A, there exists a formula $\varphi_A \in \mathrm{EMSO}^2\left(\Sigma, +1_{l_1}, +1_{l_2}\right)$ such that $L(A) = L(\varphi_A)$.*

Proof. Let $A = (B, C)$ be a 2-SS automaton with the output alphabet of B being $\Sigma' = \{l_1, \ldots, l_n\}$, $n \in \mathbb{N}$. Recall that a run of A on $\mathfrak{A}$ is a pair (ρ_b, ρ_c) where ρ_b

is a run of B and ρ_c is a run of C. We write down a formula φ_A which expresses that there is a run of A on $\mathfrak{A}$ in the following way. Let

$$\varphi_A = \exists P_{l_1} P_{l_2} \dots P_{l_n} \ (\varphi_{part}\,(P_{l_1}, \dots, P_{l_n}) \wedge \varphi_B \wedge \varphi_C)\,,$$

where (i) $\varphi_{part}\,(P_{l_1}, \dots, P_{l_n})$ says that the predicates $P_{l_1}, \dots, P_{l_n}$ form a partition of the set of all positions. These predicates act as the intermediate alphabet. (ii) φ_B is the encoding of B in $\mathrm{EMSO}^2\,(\Sigma, P_{l_1}, \dots, P_{l_n}, +1_{l_1}, +1_{l_2})$ with the predicates $P_{l_1}, \dots, P_{l_n}$ (free in φ_B) standing for the output alphabet. (iii) φ_C is the encoding of C in $\mathrm{EMSO}^2\,(P_{l_1}, \dots, P_{l_n}, +1_{l_2})$ with the predicates $P_{l_1}, \dots, P_{l_n}$ (free in φ_C) standing for the input alphabet. $\square$

6.3.2 Computing $\mathrm{msp}_{+1_{l_2}}$ from $\mathrm{msp}_{+1_{l_1}}$

In the definition of the automaton A, the transducer has access to $\mathrm{msp}_{+1_{l_1}}(\mathfrak{A})$, whereas C can only access the output of B permuted to $+1_{l_2}$. In the following, we show that it is possible for B to output a string which when permuted to $+1_{l_2}$, yields $\mathrm{msp}_{+1_{l_2}}(\mathfrak{A})$. Let $\mathfrak{A} = (A, \lambda, +1_{l_1}, +1_{l_2})$ be a 2-SS and $\mathrm{msp}_{+1_{l_1}}(\mathfrak{A}) = (a_1, b_1)(a_2, b_2) \dots (a_n, b_n)$. Let b'_i be the marking of the position i in $\mathrm{msp}_{+1_{l_2}}(\mathfrak{A})$. It is easy to verify that b'_i is a function of b_i and b_{i-1} as evidenced by the following table.

b_{i-1}	b_i	b'_i
$-$	0	0
$-$	-1	0
$-$	1	1
0	0	0

b_{i-1}	b_i	b'_i
0	1	1
1	0	0
1	1	1
0	-1	0

b_{i-1}	b_i	b'_i
-1	0	-1
-1	-1	-1
-1	1	$\perp$
1	-1	$\perp$

Note that the configurations $b_{i-1} = -1$, $b_i = 1$ and $b_{i-1} = 1$, $b_i = -1$ do not constitute a valid marking. The above table immediately gives a strategy for outputting $\mathrm{msp}_{+1_{l_2}}(\mathfrak{A})$. The automaton B always remembers the previous position's marking in its states, and computes b'_i. Once the output of B is permuted to $+1_{l_2}$, the string becomes $\mathrm{msp}_{+1_{l_2}}(\mathfrak{A})$.

6.4 Reducing $\mathrm{EMSO}^2(\Sigma, +1_{l_1}, +1_{l_2})$ to 2-SS automata

In this section we show that given an $\mathrm{FO}^2(\Sigma, +1_{l_1}, +1_{l_2})$ formula we can transform it into an equivalent 2-SS automaton. First of all, given a formula $\varphi \in \mathrm{FO}^2(\Sigma, +1_{l_1}, +1_{l_2})$ we transform it into an equivalent formula in *Scott normal form* (see Section 5.3.1). Earlier we showed that 2-SS automata are closed under renaming and intersection. Therefore it suffices to show that we can construct a 2-SS automaton for each of the formulas $\forall x \forall y\ \chi$ and $\forall x \exists y\ \psi_i$. The following two lemmas show precisely that.

In the following, a *type* is a conjunction of unary predicates or their negation. We say a formula is positive if either it is atomic or all its sub-formulas which are atomic are under the scope of an even number of negations. Similarly we say a formula is negative if all its sub-formulas which are atomic are under the scope of an odd number of negations.

Lemma 6.4.1. *Given an* $\mathrm{FO}^2(\Sigma, +1_{l_1}, +1_{l_2})$ *formula of the form* $\varphi = \forall x \forall y\ \chi$ *where* χ *is quantifier free, an equivalent* 2*-SS automaton of doubly exponential size can be constructed.*

Proof. First of all we write φ in CNF (conjunction of disjunctions, causing an exponential blowup in the size of the formula), followed by distributing the universal quantification over the conjunction, and then group them in the following way, $\bigwedge_i \forall x \forall y\ \chi_i$ where each χ_i is of the form,

$$\chi_i = \alpha(x) \vee \beta(y) \vee \epsilon(x, y) \vee \gamma(x, y) \vee \delta(x, y)$$

Above $\alpha(x)$ and $\beta(y)$ are (abusing the notation) unary types (disjunction of unary literals over the specified free variable). The formulas in the group $\epsilon(x, y)$ are $x = y$ and $x \neq y$. The formula $\gamma(x, y)$ is an order type over the order $+1_{l_1}$. By that we mean it talks about the relation between x and y with respect to the order $+1_{l_1}$. The formula $\delta(x, y)$ is an order type over the order $+1_{l_2}$. It is enough to construct a 2-SS automaton for each $\forall x \forall y\ \chi_i$ since the automata are closed under intersection. The alphabet Σ of the automata is going to be bit vectors which represent the evaluation of the unary predicates used in the formula at a given position. Hence, the size of the alphabet is exponential in the length of the formula. The automaton we construct in each case, has a constant number of states, but may have exponentially many transitions. Finally the intersection of these automata is of size doubly exponential.

Back to the reduction, some conjuncts may not have a $\delta(x, y)$ formula, some may not have $\gamma(x, y)$, some may not have both. We observe that in each of these cases, the formula χ_i talks about a regular property over one linear order by Büchi-Elgot-Trakhtenbrot theorem which says that languages accepted by finite state automata are precisely the languages definable in MSO $(\Sigma, <, +1)$, monadic second order logic over words. For instance if $\gamma(x, y)$ is absent it is a regular property over the order $+1_{l_2}$. And hence we can construct a finite state automaton running over $+1_{l_2}$, which can be converted to a 2-SS automaton easily as described in Lemma 6.3.4. If both γ and δ are absent we can verify the property on either of the orders.

Therefore we restrict our attention to those χ_i, where both γ and δ are present. Going back, γ is a disjunction of formulas from the set $O_{+1_{l_1}}$ and δ is a disjunction of formulas from the set $O_{+1_{l_2}}$, where

$$O_{+1_{l_1}} = \{+1_{l_1}(x, y), \neg{+1_{l_1}}(x, y), +1_{l_1}(y, x), \neg{+1_{l_1}}(y, x)\}$$

$$O_{+1_{l_2}} = \{+1_{l_2}(x, y), \neg{+1_{l_2}}(x, y), +1_{l_2}(y, x), \neg{+1_{l_2}}(y, x)\}$$

Suppose $\epsilon(x, y) \equiv x = y \vee x \neq y$. In this case, the formula is tautology hence we construct a 2-SS automaton which accepts all 2-SS's.

Suppose $\epsilon(x, y) \equiv x \neq y$. In this case we can rewrite χ_i as $\chi_i = (\alpha'(x) \wedge \beta'(y)) \rightarrow (x \neq y \vee \gamma(x, y) \vee \delta(x, y))$. Consider the case when $\gamma(x, y)$ and $\delta(x, y)$ are positive. In this case, whenever $\gamma(x, y) \vee \delta(x, y)$ is true, then $x \neq y$ is also true. Therefore the formula reduces to, $\chi_i = (\alpha'(x) \wedge \beta'(y)) \rightarrow (x \neq y)$ which is regular. If $\gamma(x, y)$ and $\delta(x, y)$ are not positive, one of them contains a negative formula. All negative formulas in $O_{+1_{l_1}}$ and $O_{+1_{l_2}}$ obey the following equivalence, $\varphi \equiv \varphi \vee x = y$, for example, $\neg{+1_{l_1}}(x, y) \equiv \neg{+1_{l_1}}(x, y) \vee x = y$. Therefore, χ_i can be rewritten as,

$$(\alpha'(x) \wedge \beta'(y)) \rightarrow (x \neq y \vee x = y \vee \gamma(x, y) \vee \delta(x, y))$$

which is always true. Therefore we construct a 2-SS automaton which accepts all

2-SS's.

Suppose $\epsilon(x, y) \equiv x = y$. We can rewrite χ_i in either of the following two forms,

$$\chi_i = (\alpha'(x) \wedge \beta'(y) \wedge x \neq y \wedge \gamma'(x, y)) \to \delta(x, y)$$

$$\chi_i = (\alpha'(x) \wedge \beta'(y) \wedge x \neq y \wedge \delta'(x, y)) \to \gamma(x, y)$$

where $\alpha', \beta', \gamma', \delta'$ are the negations of $\alpha, \beta, \gamma, \delta$ respectively. Note that the negations are conjunctive formulas. If δ' or γ' contains a positive formula, we choose the corresponding form. When both of them contain a positive formula we choose arbitrarily one. Suppose $\gamma'(x, y)$ contains a positive formula, in this case, we choose the following form.

$$\chi_i = (\alpha'(x) \wedge \beta'(y) \wedge x \neq y \wedge \gamma'(x, y)) \to \delta(x, y)$$

The only satisfiable $\gamma'(x, y)$ can be the following, $+1_{l_1}(x, y)$, $+1_{l_1}(y, x)$, $+1_{l_1}(x, y) \wedge \neg{+1_{l_1}}(y, x)$ and $+1_{l_1}(y, x) \wedge \neg{+1_{l_1}}(x, y)$. The formula $+1_{l_1}(x, y) \wedge \neg{+1_{l_1}}(y, x)$ reduces to $+1_{l_1}(x, y)$, similarly $+1_{l_1}(y, x) \wedge \neg{+1_{l_1}}(x, y)$ reduces to $+1_{l_1}(y, x)$. This leaves us with two possible cases for $\gamma'(x, y)$ which are $+1_{l_1}(x, y)$ and $+1_{l_1}(y, x)$. The formula $\delta(x, y)$ is a disjunction of formulas from the set $O_{+1_{l_2}}$. We claim that in this case, the automaton can verify the formula χ_i, by looking at the marked string projection to the order $+1_{l_1}$. For example when α' holds at x, β' holds at y, γ' is $+1_{l_1}(x, y)$ and when δ is $+1_{l_2}(x, y)$ the automaton B checks the marking of x is 1. Similarly, when δ is $+1_{l_2}(y, x)$, $\neg{+1_{l_2}}(x, y)$, $\neg{+1_{l_2}}(y, x)$ the marking at x has to be respectively -1, 0 or -1, 0 or 1. If δ is a disjunction, the automaton can guess one of the disjuncts and verify it. The case when γ' is $+1_{l_1}(y, x)$ is similar, instead of looking at the marking of x, the automaton B checks the marking of y.

When δ' contains a positive formula, we choose the form,

$$\chi_i = (\alpha'(x) \wedge \beta'(y) \wedge x \neq y \wedge \delta'(x, y)) \to \gamma(x, y)$$

The reasoning is similar, we can reduce δ' to two cases, $+1_{l_2}(x, y)$ and $+1_{l_2}(y, x)$. The formula γ is a disjunction of formulas from the set $O_{+1_{l_1}}$. In this case the automaton verifies the formula by checking the marked string projection to the order $+1_{l_2}$. We showed earlier that this can be done by a 2-SS automaton.

The only remaining case is when γ' and δ' both do not contain a positive formula. Therefore both γ' and δ' are negative formulas, therefore γ and δ are positive formulas. We rewrite χ_i in the following form,

$$\chi_i = (\alpha'(x) \wedge \beta'(y) \wedge x \neq y) \rightarrow (\gamma(x,y) \vee \delta(x,y))$$

The formula says the following. Whenever α' holds at x and β' holds at y and x, y are distinct then either they are neighbours in $+1_{l_1}$ as dictated by γ or neighbours in $+1_{l_2}$ as dictated by δ. If there is no α' in the word there can be any number of β'. Similarly there can be any number of α' if there is no β' occurring in the word. The automaton B can guess both these cases and verify them easily. So henceforth we assume that there is at least one α' and β' present in the word. In which case as we show below, the number of α' and β' are bounded. We do a case analysis.

Consider the case when $\gamma \equiv +1_{l_1}(x,y)$ and $\delta \equiv +1_{l_2}(x,y)$. Let's say β' occurs at position x, then there can be at most three α', one α' at the predecessor of x in $+1_{l_1}$, one α' at the predecessor of x in $+1_{l_2}$ and one α' at x. (Similarly if there is a β' there can be at most three α', though we do not use this fact in the following construction). Firstly, the transducer B guesses the number of α occurring in the word, which is let's say k, $1 \leq k \leq 3$ and labels them as $\alpha_1, \ldots, \alpha_k$ in the output. Let the $\alpha_1 \ldots \alpha_k$ occurs at the positions $x_1, \ldots, x_k$ respectively. For every $\beta(y)$ occurring in the word the automaton labels the position with a vector $(b_{\alpha_1}(y), \ldots, b_{\alpha_k}(y))$ where the vector is defined in the following way.

$$b_{\alpha_i}(y) = \begin{cases} 1 & \text{if } x_i = y, \\ 1 & \text{if } x_i \neq y,\ \mathfrak{A}, x_i, y \models \gamma(x,y), \\ 0 & \text{if } x_i \neq y,\ \mathfrak{A}, x_i, y \not\models \gamma(x,y). \end{cases}$$

This can be done because for every $\beta(y)$, we only need to know the α-s which are neighbours of y in $+1_{l_1}$. Since the number of α-s is bounded, by making use of finite memory and nondeterminism the automaton B will be able to determine, which α-s are neighbours of each $\beta(y)$. The automaton C does the following when it runs over the output of B. (1) For every $\beta(y)$ occurring in the word it computes a vector $\left(b'_{\alpha_1}(y), \ldots, b'_{\alpha_k}(y)\right)$ where the vector is defined in the following way.

$$b'_{\alpha_i}(y) = \begin{cases} 1 & \text{if } x_i = y, \\ 1 & \text{if } x_i \neq y,\ \mathfrak{A}, x_i, y \models \delta(x, y), \\ 0 & \text{if } x_i \neq y,\ \mathfrak{A}, x_i, y \not\models \delta(x, y). \end{cases}$$

The automaton C depends on the labellings of the α-s by B to compute this. (2) For each $\beta(y)$ the automaton verifies that for every $1 \leq i \leq k$ at least one of $b_{\alpha_i}(y)$ or $b'_{\alpha_i}(y)$ is one. This step is easily done by accessing the tagged vector of each $\beta(y)$.

In the cases where $\gamma \vee \delta$ is one of $+1_{l_1}(y, x) \vee +1_{l_2}(x, y)$, $+1_{l_1}(x, y) \vee +1_{l_2}(y, x)$, $+1_{l_1}(y, x) \vee +1_{l_2}(y, x)$, the number of α-s is bounded by three. When $\gamma \vee \delta$ is $+1_{l_1}(x, y) \vee +1_{l_1}(y, x) \vee +1_{l_2}(x, y) \vee +1_{l_2}(y, x)$, the number of α-s is bounded by five. In all other cases the number of α-s is bounded by four. In all the above cases, the construction is similar.

This completes the proof. □

Lemma 6.4.2. *For each* $\mathrm{FO}^2(\Sigma, +1_{l_1}, +1_{l_2})$ *formula of the form* $\forall x \exists y\ \psi$ *where* ψ *is quantifier free, an equivalent* 2*-SS automaton of doubly exponential size can be constructed.*

Proof. First of all we note that, ψ_i can be written (using the truth table for ψ_i) as an exponential size conjunction of disjunctions of the form

$$\forall x \exists y \bigwedge_i \bigvee_j (\alpha_i(x) \rightarrow \theta_{ij}(x, y))$$

where α_i enumerates through all possible maximal types, that is $\bigvee_i(\alpha_i(x))$ is a tautology and $\neg(\alpha_i(x) \wedge \alpha_j(x))$ for all $i \neq j$. The formula θ_{ij} is either *false* or of the form,

$$\beta_{ij}(y) \wedge \epsilon_{ij}(x, y) \wedge \gamma_{ij}(x, y) \wedge \delta_{ij}(x, y)$$

where, β_{ij} is a type, ϵ_{ij} is one of $x = y$, $x \neq y$, γ_{ij} is one of $+1_{l_1}(x, y)$, $+1_{l_1}(y, x)$, $\neg{+1_{l_1}}(x, y)$, $\neg{+1_{l_1}}(y, x)$, $\neg{+1_{l_1}}(x, y) \wedge \neg{+1_{l_1}}(y, x)$ and δ_{ij} is one of $+1_{l_2}(x, y)$, $+1_{l_2}(y, x)$, $\neg{+1_{l_2}}(x, y)$, $\neg{+1_{l_2}}(y, x)$, $\neg{+1_{l_2}}(x, y) \wedge \neg{+1_{l_2}}(y, x)$.

We notice that the premises occurring in distinct conjuncts are distinct (and mutually exclusive). Hence it is possible to distribute the $\forall x \exists y$ over the conjunction. The resulting formula is of the form

$$\wedge_i \forall x \exists y \ \vee_j (\alpha_i(x) \to \theta_{ij}(x,y))$$

We eliminate the disjunction by adding to every disjunct a new unary predicate $\Lambda_{ij}(x)$ (these predicates are chosen to be distinct for each ψ_i) which denotes that at the position x, the j-th disjunct is witnessing α_i. We can rewrite every conjunct in the above formula as

$$\begin{aligned} &\exists \Lambda_{i1}\Lambda_{i2}\ldots\Lambda_{ik} \, (\forall x \vee_j \Lambda_{ij}(x)) \\ &\qquad\qquad \wedge \bigwedge_j \forall x \exists y \ \ ((\alpha_i(x) \wedge \Lambda_{ij}(x)) \to \theta_{ij}(x,y)) \end{aligned}$$

A 2-SS automaton can guess the predicates Λ_{ij} nondeterministically and verify them. Hence our job is complete once we describe how to construct a 2-SS automaton for each formula of the form $\forall x \exists y \ \ (\alpha(x) \to \theta_{ij}(x,y))$. If the consequent is false, the language is regular. So we concentrate on the cases where the consequent is satisfiable.

$$\forall x \exists y \ \ (\alpha(x) \to (\beta(y) \wedge \epsilon(x,y) \wedge \gamma(x,y) \wedge \delta(x,y)))$$

We do a case analysis. If $\epsilon(x,y) \equiv x = y$, the language is regular. Hence now onwards we fix ϵ to be $x \neq y$. As in the previous proof, we have two cases, when γ or δ contains a positive formula and when they do not. Suppose γ contains a positive formula, in this case γ reduces to either $+1_{l_1}(x,y)$ or $+1_{l_1}(y,x)$.

Let γ be $+1_{l_1}(x,y)$. The formula δ reduces to one of $+1_{l_2}(x,y)$, $+1_{l_2}(y,x)$, $\neg{+1_{l_2}}(x,y)$, $\neg{+1_{l_2}}(y,x)$, $\neg{+1_{l_2}}(x,y) \wedge \neg{+1_{l_2}}(y,x)$. The automaton B can check these cases by verifying that y is the successor of x in $+1_{l_1}$, $\beta(y)$ holds and the marking at x is consistent with δ. The case for $\gamma \equiv +1_{l_1}(y,x)$ is similar.

When δ contains a positive formula the construction is similar, except that now the automaton C verifies the formula.

The only remaining case is when γ and δ both do not contain a positive formula.

Consider the case when $\gamma \equiv \neg{+1}_{l_1}(x,y)$ and $\delta \equiv \neg{+1}_{l_2}(x,y)$. The formula says that if there is an α at x there should be a witness y with β holding there, such that y is not a successor of x in both the orders. Notice that if there are at least four β-s occurring in the word we will be able to find a witness for any α, since any position can have at most two β-s as its successors and one β at the position itself, hence the fourth β will witness it. The automaton guesses whether the word contains at least four β-s and verifies it, in which case the formula is taken care of. Suppose the automaton guesses that the word contains fewer than four β-s. In this case, the automaton guesses that there are exactly $k, 0 \leq k \leq 3$ positions satisfying β and verifies it. If $k = 0$ there should not be any positions with α, and this case is regular. Otherwise the automaton B labels the β-s as $\beta_1, \ldots \beta_k$ and outputs them. Let the positions with β be $y_1 \ldots y_k$. For each $\alpha(x)$ occurring in the word the automaton B tags the position x with a vector $(b_{\beta_1}(x), \ldots b_{\beta_k}(x))$ where the vector is defined in the following way:

$$b_{\beta_i}(x) = \begin{cases} 0 & \text{if } x = y, \\ 1 & \text{if } x \neq y_i,\ \mathfrak{A}, x, y_i \models \gamma(x,y), \\ 0 & \text{if } x \neq y_i,\ \mathfrak{A}, x, y_i \not\models \gamma(x,y). \end{cases}$$

For determining the vector the automaton needs to know which β is the successor of $\alpha(x)$ in ${+1}_{l_1}$, if there is one. Since the number of β-s is bounded, by making use of finite memory and non-determinism the automaton B will be able to determine this. The automaton C does the following when it runs over the output of B. (1) For every $\alpha(x)$ occurring in the word it computes a vector $\left(b'_{\beta_1}(x), \ldots, b'_{\beta_k}(x)\right)$ where the vector is defined in the following way:

$$b'_{\beta_i}(x) = \begin{cases} 0 & \text{if } x = y_i, \\ 1 & \text{if } x \neq y_i,\ \mathfrak{A}, x, y_i \models \delta(x,y), \\ 0 & \text{if } x \neq y_i,\ \mathfrak{A}, x, y_i \not\models \delta(x,y). \end{cases}$$

The automaton C depends on the labellings of the β-s by B to compute this. (2) For each $\alpha(x)$ the automaton verifies that there is an $1 \leq i \leq k$ such that both $b_{\beta_i}(x)$ and $b'_{\beta_i}(x)$ are one. This step is easily done by accessing the tagged vector of each $\alpha(x)$.

In the cases where $\gamma \wedge \delta$ is one of $\neg{+1}_{l_1}(y,x) \wedge \neg{+1}_{l_2}(x,y)$, $\neg{+1}_{l_1}(x,y) \wedge$

$\neg{+1}_{l_2}(y,x)$, $\neg{+1}_{l_1}(y,x) \wedge \neg{+1}_{l_2}(y,x)$, the sufficient number of β-s is three. When $\gamma \wedge \delta$ is $\neg{+1}_{l_1}(x,y) \wedge \neg{+1}_{l_1}(y,x) \wedge \neg{+1}_{l_2}(x,y) \wedge \neg{+1}_{l_2}(y,x)$, the sufficient number of β-s is five. In all other cases the sufficient number of β-s is four. In all the above cases, the construction is similar.

This completes the proof. □

The result from this section along with Proposition 6.3.9 imply the following,

Proposition 6.4.3. *2-SS automata and* $\mathrm{EMSO}^2(\Sigma, {+1}_{l_1}, {+1}_{l_2})$ *are equivalent in terms of expressiveness.*

Hence checking satisfiability of a formula φ in the logic reduces to checking non-emptiness of the corresponding automaton A_φ.

6.5 Decidability of 2-SS automata

In this section we prove that checking emptiness of a 2-SS automaton is decidable. A permutation π over a set S, is a bijective map from S to S. Let $w = ([n], \lambda, {+1}_l)$ be a word and let π be a permutation over $[n]$. We call $\pi(w) = ([n], \pi^{-1} \circ \lambda, {+1}_l)$ a permutation of w. We define *perm*(w) as the set of all permutations of w. For a language L, let *perm*(L) be the set of words that are the permutations of the words in L.

For example if $L = (abc)^*$ then *perm*$(L) = \{w \mid w \in \{a,b,c\}^*, \#_a(w) = \#_b(w) = \#_c(w)\}$. Notice that *perm*$(L)$ can be non-regular even if L is regular. In the previous case, it is not even context-free. But it is the case that for a regular language L over a two letter alphabet, *perm*(L) is context-free.

Proposition 6.5.1. *If L is regular then perm(L) is accepted by a multicounter automaton.*

Proof. The statement of this proposition is clear, since the Parikh image of any regular language is semi-linear. However we present a proof below, which will be extended later. The idea of the proof is to construct a multicounter automaton M_A, given a finite state automaton A, such that *perm*$(L(A)) = L(M_A)$. Though there are many ways to achieve this, in the following, we describe a construction which we can extend later.

Let L be a regular language, then there is a finite state automaton $A = (Q, \Sigma, \Delta, I, F)$ such that $L(A) = L$. Given a word $w = ([n], \lambda, +1_l)$, it is in $L(A)$ if there is a run $\rho = \delta_1 \ldots \delta_n \in \Delta^*$ of A on w such that ρ is accepting.

The idea of our construction is the following. Given a word w, the multicounter automaton assigns a transition from Δ to each letter of the word. The automaton then checks if the those partial runs can be joined arbitrarily to create a successful run. The counters of the multicounter automaton is given by the set $C = (Q \times Q)$. Now onwards we refer to the partial runs as blocks. At any point during the computation the following invariant is kept: *If the counter (p, q) has value k then there are exactly k blocks corresponding to partial runs from p to q.* Initially all the counters are zero and the invariant is satisfied trivially.

When the automaton encounters a letter a at position i, first of all it guesses a transition $(p, a, q) \in \Delta$. Now, the following scenarios can occur.

(t_1) The automaton guesses two blocks (p_1, q_1) and (p_2, q_2) such that $q_1 = p$ and $p_2 = q$. The counters corresponding to the blocks (p_1, q_1) and (p_2, q_2) are decremented by one and the counter corresponding to (p_1, q_2) is incremented by one. Note that the update of counters amounts to merging two blocks to the left and right of the current transition.

(t_2) The automaton guesses the right block (p_2, q_2) such that $p_2 = q$. The counter corresponding to (p_2, q_2) is decremented by one and the counter corresponding to (p, q_2) is incremented by one. In this case the left block will be merged to the current transition when it appears in the future.

(t_3) The automaton guesses the left block (p_1, q_1) such that $q_1 = p$. The counter corresponding to (p_1, q_1) is decremented by one and the counter corresponding to (p_1, q) is incremented by one. In this case the right block will be merged to the current transition when it appears in the future.

(t_4) The automaton simply increments the counter corresponding to (p, q) by one. In this case the transition is returned to the counters as an individual block.

Note that in all the above cases our invariant holds. Finally, at the end of the simulation the multicounter automaton accepts if: (1) a counter corresponding to $(p, q) \in I \times F$ is one, (2) all other counters empty.

We need to show that if $w \in perm(L(A))$ if and only if M_A has a successful run on w. The right to left direction is guaranteed by the invariant. Since at the end of the simulation M_A accepts if there is a successful run of A on w.

For the left to right direction, assume that the word w of length n has a permutation $\pi : [n] \to [n]$ such that $\pi(w) \in L(A)$. Let $\rho = \delta_1 \ldots \delta_n$ be an accepting run of $\pi(w)$ on A. For $1 \leq i \leq n$, we refer to the set $S = \{\pi(1), \ldots, \pi(i)\}$ as the partial permutation corresponding to position i. The set $s \subseteq S$ is a maximal segment in S if: (1) the set $s = \{i, i+1, \ldots, j\}$ for some $i \leq j$ (2) $i - 1$ and $j + 1$ are not in S. Given any partial permutation it can be partitioned into a number of maximal segments. The partial run corresponding to the segment s is $\delta_i \ldots \delta_j$. The block corresponding to the segment s is the pair (p, q) where p is the start state of δ_i and q is the end state of δ_j. For each position i we define the counter configuration $h_i : C \to \mathbb{N}$ as;

$$h_i((p, q)) = \# \text{ of maximal segments of } \{\pi(1), \ldots, \pi(i)\} \text{ whose blocks are } (p, q)$$

Next we describe a successful run of the multicounter automaton on w. The automaton chooses the transition $\delta_{\pi(i)}$ at position i. We claim that at position i the automaton can reach the counter configuration h_i. We prove it using induction on i. Initially all the counters are empty and the condition is trivially satisfied. For the inductive step, assume that after position i the automaton reached the configuration h_i.

At position $i + 1$ the automaton selects the transition $\delta_{\pi(i+1)} = (p, a, q)$. Let $S = \{\pi(1), \ldots, \pi(i)\}$ be the partial permutation corresponding to position i and let the maximal segments of S be $M = \{s_1, \ldots, s_k\}$, $1 \leq k \leq i$.

If $\pi\left(\pi^{-1}(i+1) - 1\right) < i + 1$ and $\pi\left(\pi^{-1}(i+1) + 1\right) < i + 1$ we observe that there are two maximal segments s_l, s_r in S whose blocks are (p_1, p) and (q, q_2) for some $p_1, q_2 \in Q$. Hence, by induction hypothesis the values of these counters are greater than zero. The automaton performs the action t_1. The maximal segments of $S \cup \{\pi(i+1)\}$ are $(M - \{s_l, s_r\}) \cup \{s_l \cup s_r \cup \{\pi(i+1)\}\}$. The count of blocks for segments in $(M - \{s_l, s_r\})$ remains unchanged. While the number of blocks corresponding to (p_1, p) and (q, q_2) decreases by one and the number of blocks corresponding to block (p_1, q_2) increases by one. This is reflected by the counter update in the action t_1.

If $\pi(\pi^{-1}(i+1) - 1) < i+1$ and $\pi(\pi^{-1}(i+1) + 1) > i+1$ there is maximal segment s_l in S with block (p_1, p) for some $p_1 \in Q$. The counter value of (p_1, p) is greater than zero by induction hypothesis. The automaton takes the step t_3 and the counter values is updated. The updated counter value reflects the block counts of maximal segments in $S \cup \{\pi(i+1)\}$ which is the set $(M - \{s_l\}) \cup \{s_l \cup \{\pi(i+1)\}\}$. The scenario when $\pi(\pi^{-1}(i+1) - 1) > i+1$ and $\pi(\pi^{-1}(i+1) + 1) < i+1$ is symmetric and in this case the automaton performs the action t_2.

When $\pi(\pi^{-1}(i+1) - 1) > i+1$ and $\pi(\pi^{-1}(i+1) + 1) > i+1$, the maximal segments of $S \cup \{\pi(i+1)\}$ is the set $M \cup \{\{\pi(i+1)\}\}$ and the automaton performs the action t_4.

At the end of the run the multicounter automaton has a single maximal segment with block (p, q) for some $p \in I$ and $q \in F$. Hence, by the above claim all counters except (p, q) are zero and the counter corresponding to (p, q) is one and the automaton accepts. $\square$

The above theorem shows that if we did not have the marking on the words, the decidability follows immediately. Since, given the 2-SS automaton $A = (B, C)$ we could construct an ϵ-free multicounter automaton which accepts $perm(L(C))$ and intersect it with the finite state transducer B (in such a way that output of B is supplied as the input of the multicounter automaton) and check the emptiness of the whole system. Next we show how to adapt this technique to the case of marked words.

Let $w = (a_1, b_1) \dots (a_n, b_n) \in (\Sigma \times \{1, 0, -1\})^*$ be a marked word. We say u is a -1-factor of w if u is a factor (subword defined by adjacent positions) of w and all the letters in u have the marking -1 except the last position which is marked by 0. Similarly u is a 1-factor of w if u is a factor of w such that all the letters in u have the marking 1 except the last position which is marked by 0. Given any marked word w, it can be factorised into $w = u_1 u_2 \dots u_k$, $k \leq n$ where each u_i is a 1-factor or -1-factor. For easiness we refer to them as factors.

Lemma 6.5.2. *Given a 2-SS automaton $\mathcal{A} = (B, C)$, there is a 2-SS automaton $\mathcal{A}' = (B', C')$ with the following properties. (1) The factors of every marked word accepted by B' has length at least two. (2) $L(\mathcal{A})$ is non-empty if and only if $L(\mathcal{A}')$ is non-empty. Moreover, $\mathcal{A}'$ can be obtained from $\mathcal{A}$ in linear time.*

Proof. Let the alphabet of $\mathcal{A}$ be Σ. We set the alphabet of $\mathcal{A}'$ as $\Sigma \cup \{\Box\}$ where $\Box$ is a dummy letter.

Let $B = (Q, \Sigma, \Sigma', \Delta, O, I, F)$. Let B_1 be $B_1 = (Q, \Sigma \cup \{\Box\}, \Sigma' \cup \{\Box\}, \Delta', O', I, F)$ where $\Delta' = \Delta \cup \{(p, (\Box, 1), p) \mid p \in Q\}$ and $O' = O \cup \{((p, (\Box, 1), p), \Box) \mid p \in Q\}$.

Let B_2 be the finite state automaton accepting the language "All factors are of length greater than one". Define B' as the intersection of B_1 and B_2.

Let C be $C = (Q_c, \Sigma', \Delta_c, I_c, F_c)$, we define the automaton C' to be $C' = (Q_c, \Sigma', \Delta_c' = \Delta_c \cup \{(p, \Box, p) \mid p \in Q\}, I_c, F_c)$.

The first claim follows from the fact that no marked words accepted by B' has marking 0 appearing in consecutive positions. This is guaranteed by the automaton B_2.

Next we show that $L(\mathcal{A})$ is non-empty if and only if $L(\mathcal{A}')$ is non-empty.

For the left-to-right direction, let $\mathfrak{A} = (A, \lambda, +1_{l_1}, +1_{l_2})$ be a 2-SS in $L(\mathcal{A})$. If $msp_{+1_{l_1}}(\mathfrak{A})$ does not have factors of length one, we are done. If it is not the case, we introduce some new elements into the structure $\mathfrak{A}$ with the label $\Box$ while preserving the relative orderings (in both the orders) of elements in A. Let F be the sets of pairs in $+1_{l_1}$ constituting factors of length one.

$$F = \{(x, y) \mid x, y \in A, +1_{l_1}(x, y), msp_{+1_{l_1}}(x) = 0, msp_{+1_{l_1}}(y) = 0\}$$

We define a new 2-SS $\mathfrak{A}' = \left(A', \lambda', +1'_{l_1}, +1'_{l_2}\right)$ where;

$$A' = A \cup \{e_{(x,y)} \mid (x, y) \in F\}$$

$$\lambda'(x) = \begin{cases} \lambda(x) & \text{if} \quad x \in A \\ \Box & \text{if} \quad \text{Otherwise} \end{cases}$$

$$+1'_{l_1} = \{(x, e_{(x,y)}), (e_{(x,y)}, y) \mid (x, y) \in F\} \cup (+1_{l_1} - F)$$

$$
\begin{aligned}
+1'_{l_2} =& \{(x,y) \mid (x,y) \in +1_{l_2}, y \notin \mathsf{Range}(F)\} \\
& \cup \{\big(z, e_{(x,y)}\big) \mid (z,y) \in +1_{l_2}, y \in \mathsf{Range}(F)\} \\
& \cup \{(e_{(x,y)}, y) \mid (x,y) \in F\}
\end{aligned}
$$

We claim that $\mathfrak{A}'$ has a successful run on $\mathcal{A}'$. Observe that,

$$msp_{+1'_{l_1}}(\mathfrak{A}') = u_1(\square,1)u_2(\square,1)\ldots(\square,1)u_k$$

where $u_1u_2\ldots u_k = msp_{+1_{l_1}}(\mathfrak{A})$ and each u_j is a factor of $msp_{+1_{l_1}}(\mathfrak{A})$. Let $\rho = (\rho_b, \rho_c)$ an accepting run of $\mathcal{A}$ on $\mathfrak{A}$. The run ρ_b of B on $u_1u_2\ldots u_k$ can be grouped as $\rho_b = \rho_1\rho_2\ldots\rho_k$ where each ρ_i is a partial run on u_i. From the definition of B' it follows that $\rho'_b = \rho_1(p_1,(\square,1),p_1)\rho_2(p_2,(\square,1),p_2)\ldots(p_k,(\square,1),p_l)\rho_k$ is an accepting run of B' on $msp_{+1'_{l_1}}(\mathfrak{A}')$. Here we suppressed the fact that the automaton B' is an intersection of B_1 and B_2 for the sake of simplicity. Since $msp_{+1'_{l_1}}(\mathfrak{A}')$ is in the language of B_2 it is not an impediment but a thorny technical detail.

Similarly the output word of B' is of the form $w' = v_1\square v_2\square\ldots\square v_k$ where $v_1v_2\ldots v_k$ is the output word of B. By a similar argument it follows that w' is accepted by the automaton C.

For the right-to-left direction, assume that $L(\mathcal{A}')$ is non-empty. Hence there exists a 2-SS $\mathfrak{A}' = \big(A', \lambda', +1'_{l_1}, +1'_{l_2}\big)$ in $L(\mathcal{A}')$. If no position in $\mathfrak{A}'$ is labelled by $\square$ we are done since $\mathfrak{A}'$ has an accepting run on $\mathcal{A}$. Otherwise we define $\mathfrak{A}$ to be the structure $\mathfrak{A} = (A, \lambda, +1_{l_1}, +1_{l_2})$ where $A = \{x \in A' \mid \lambda(x) \neq \square\}$. λ' as λ restricted to A and $+1_{l_1}, +1_{l_2}$ as,

$$+1_{l_1} = \{(x,y) \mid (x,y) \in +1'_{l_1}, \forall z, x \leq_{l_1} z \leq_{l_2} y \implies \lambda(z) = \square\}$$

$$+1_{l_z} = \{(x,y) \mid (x,y) \in +1'_{l_2}, \forall z, x \leq_{l_2} z \leq_{l_2} y \implies \lambda(z) = \square\}$$

From $\mathfrak{A}'$ we remove the positions carrying the label $\square$ while keeping the relative ordering (on both the orders) of remaining positions. Most importantly this does not change the marking $msp_{+1_{l_1}}$ of positions in A.

Notice that $msp_{+1_{l_1}}(\mathfrak{A})$ is the subword of $msp_{+1'_{l_1}}(\mathfrak{A}')$ obtained by removing $(\Box, 1)$. Hence the accepting run ρ'_b of B' can be converted to an accepting run of B on $msp_{+1_{l_1}}(\mathfrak{A})$ by removing the transitions on $(\Box, 1)$. Similarly the output word w' of B is the subword of output word w of B by removing $\Box$. By definition of C', we can infer that C has an accepting run on w'. $\Box$

We define the notion of a marked permutation. Given a marked word $w = (a_1, b_1)(a_2, b_2) \dots (a_n, b_n)$ where $b_i \in \{-1, 0, 1\}$, we say a permutation $\pi : [n] \to [n]$ defines a marked permutation of w iff (1) for every i, if $b_i = 1$ then $\pi(i) + 1 = \pi(i+1)$. Note that for the last position b_i is always zero. (2) for every i, if $b_i = -1$ then $\pi(i) = \pi(i+1) + 1$. (3) Whenever $b_i = 0$ and i is not the last position then $\pi(i) + 1 \neq \pi(i+1)$ and $\pi(i) \neq \pi(i+1) + 1$. We call $\pi(w)$ as the marked permutation of w. Given a word w over Σ , by $mperm(w)$ we mean all marked words w' such that w is a marked permutation of w and for a language L of words over Σ, by $mperm(L)$ we mean the set of marked words w' such that w' has a marked permutation w which is in L.

Lemma 6.5.3. *If L is regular language, then the set of all words in $mperm(L)$ with factors of length at least two is accepted by a multicounter automaton.*

Proof. The multicounter automaton checks if positions of w can be permuted (satisfying the marking) to obtain a word in L. Let $A = (Q, \Sigma, \Delta, I, F)$ be a finite state automaton accepting the language L.

Let $w = u_1 u_2 \dots u_k$ be a marked word where each u_i is a factor. Then $w \in mperm(L)$ if there is a permutation $i_1, i_2, \dots, i_k$ of $1, 2, \dots, k$ such that the word $u^*_{i_1} u^*_{i_2} \dots u^*_{i_k} \in L$ where $u^*_i = str(u_i)$ if u_i is a 1-factor and $u^*_i = (str(u_i))^r$ if u_i is a -1-factor such that if u_i and u_{i+1} are 1-factors in the permutation u^*_i is not followed by u^*_{i+1} and if u_i and u_{i+1} are -1-factors in the permutation u^*_i is not followed by u^*_{i+1}. The last condition ensures that the permutation obeys the marking.

Stating the above in terms of the run of A, the word w belongs to $mperm(L)$ if there exists pairs $(p_1, q_1), \dots, (p_k, q_k)$ such that;

- For all i, if u_i is a 1-factor then $str(u_i)$ has run from p_i to q_i. For all i, if u_i is a -1-factor then $(str(u_i))^r$ has run from p_i to q_i.

- There is a permutation $\pi = (p_{i_1}, q_{i_1}), \ldots, (p_{i_k}, q_{i_k})$ of $(p_1, q_1), \ldots, (p_k, q_k)$ such that
  - $p_{i_1} \in I$, $q_{i_k} \in F$ and forall i_j, $q_{i_j} = p_{i_{j+1}}$.
  - For all i, if u_i and u_{i+1} are both 1-factors then in the permutation π, the pair (p_i, q_i) is not followed by (p_{i+1}, q_{i+1}).
  - For all i, if u_i and u_{i+1} are both -1-factors then in the permutation π, the pair (p_{i+1}, q_{i+1}) is not followed by (p_i, q_i).

We construct a multicounter automaton which checks the above condition. The automaton has counters from the set $C = (Q \times Q)$ to store the count of the blocks seen so far. Given a marked word $w = u_1 u_2 \ldots u_k$ where each u_j is a factor of w, as in the proof of Lemma 6.5.1, the multicounter automaton works as follows. While reading each factor u_{i-1} it guesses a pair (p_{i-1}, q_{i-1}) and verifies that u_{i-1} has a partial run starting in the state p_{i-1} and ending in state q_{i-1}. This block may then combined on the left and right with partial runs stored in the counters resulting in block (p', q') (Shortly, we will describe this step in detail). Finally the block (p', q') is returned to the counters. To ensure that a run is consistent with the marking, the automaton remembers in its state the following information: Whether the factor u_{i-1} is a 1-factor or a -1-factor, whether the block was merged to the left, whether the block was merged to the right, the resulting block (p', q').

Next we describe the construction in detail. Assume that the automaton is reading u_i and verified that it corresponds to a block (p, q). The following scenarios can occur,

(t_1) The automaton guesses two blocks (p_1, q_1) and (p_2, q_2) such that $q_1 = p_i$ and $p_2 = q_i$. If u_{i-1} is a 1-factor, $(p_1, q_1) = (p', q')$ and the block (p_{i-1}, q_{i-}) was not merged to the right then the automaton verifies that the counter corresponding to (p', q') is at least two. We use the fact that factors are of length at least two here. Since the factor is of length at least two, we just have to make sure that the block corresponding to u_{i-1} is not merged to the right, while it is not a problem to merge it to the left. If u_{i-1} is a -1-factor, $(p_2, q_2) = (p', q')$ and the block (p_{i-1}, q_{i-}) was not merged to the left then the automaton verifies that the counter corresponding to (p', q') is at least two.

This is to ensure that there is a block (p', q') which does not correspond to partial run over u_{i-1} violating the consistency condition. Here also it is not a problem to merge the block to the right because the factors are of length at least two. The counters corresponding to the blocks (p_1, q_1) and (p_2, q_2) are decremented by one and the counter corresponding to (p_1, q_2) is incremented by one. Note that the update of counters amounts to merging two blocks to the left and right of the current block.

(t_2) The automaton guesses the right block (p_2, q_2) such that $p_2 = q$. If u_{i-1} is a -1-factor, $(p_1, q_1) = (p', q')$ and the block (p_{i-1}, q_{i-}) was not merged to the left then the automaton verifies that the counter corresponding to (p', q') is at least two. This is to ensure that there is a block (p', q') which does not correspond to partial run over u_{i-1}. The counter corresponding to (p_2, q_2) is decremented by one and the counter corresponding to (p, q_2) is incremented by one. In this case the left block will be merged to the current block when it appears in the future.

(t_3) The automaton guesses the left block (p_1, q_1) such that $q_1 = p$. If u_{i-1} is a 1-factor, $(p_1, q_1) = (p', q')$ and the block (p_{i-1}, q_{i-}) was not merged to the right then the automaton verifies that the counter corresponding to (p', q') is at least two. This is to ensure that there is a block (p', q') which does not correspond to partial run over u_{i-1}. The counter corresponding to (p_1, q_1) is decremented by one and the counter corresponding to (p_1, q) is incremented by one. In this case the right block will be merged to the current block when it appears in the future.

(t_4) The automaton simply increments the counter counter corresponding to (p, q) by one. In this case the block (p, q) is returned to the counters.

Finally at the end of the simulation if all counters are empty except a counter $(p, q) \in (I \times F)$ the multicounter automaton accepts.

To show that if the multicounter automaton accepts a marked word w with factors of size at least two then w is in $mperm(L)$, we proceed as in the case of Proposition 6.5.1. The invariant that during the run a counter (p, q) has value k if and only there are k partial runs from p to q still holds. Also, the consistency condition incorporated into the actions of the multicounter automaton allows us to find a block whenever there is a conflict in merging two blocks.

Similarly, to show that if w has factors of length at least two and $w \in mperm(L)$ then the multicounter automaton has an accepting run on w, the proof follows the same reasoning as in the proof of Proposition 6.5.1. We consider factors as individual positions. Consider the following scenario. The automaton read u_i and verified the partial run (p, q). The left neighbour of the block u_i precedes u_i and corresponds to the block (p', q'). At the same time the factor u_{i-1} is a 1-factor with corresponding the block (p', q') and was not merged to the right. In this case there exists yet another maximal segment with block (p', q'). Hence the value of the counter (p', q') is at least two and the automaton executes action t_2. The arguments for t_3, t_1 is symmetric. □

Theorem 6.5.4. *Emptiness checking of* 2*-SS automata is decidable.*

Proof. Given the 2-SS automaton $A = (B, C)$, we first construct the 2-SS $(A' = (B', C')$ such that the marked words accepted by B' have factors of length at least two (using Lemma 6.5.2). Construct a multicounter automaton $P_{C'}$ which accepts $mperm\,(L\,(C'))$. We take the intersection of the transducer B' and $P_{C'}$ such a way that the output of B' is supplied as the input of $P_{C'}$. Finally we check the emptiness of the resulting automaton. □

6.5.1 Remarks

Theorem 6.5.4 along with Proposition 6.4.3 yield a decision procedure for testing finite satisfiability of $\mathrm{FO}^2\,(\Sigma, +1_{l_1}, +1_{l_2})$ formulas. Given $\varphi \in \mathrm{FO}^2\,(\Sigma, +1_{l_1}, +1_{l_2})$ we construct a 2-SS automaton A_φ accepting models of φ and check the emptiness of A_φ. It follows that;

Theorem 6.5.5. *Finite satisfiability of* $\mathrm{FO}^2\,(\Sigma, +1_{l_1}, +1_{l_2})$ *is decidable.*

We want to draw the attention to why the decidability proof does not generalize to $\mathrm{FO}^2\,(\Sigma, +1_{l_1}, +1_{l_2}, +1_{l_3})$. The reason is that we relied on $msp_{+1_{l_1}}$ to compute $msp_{+1_{l_2}}$. This step is not possible in the case of three successor relations. This can be however overcome by providing the component automata (each running on

$+1_{l_1}, +1_{l_2}$ and $+1_{l_3}$) with their own marked string projection as we see in the next section.

6.6 n-Successor Structures

Next we try to generalize the constructions seen before to the case of structures with n successor relations.

A n-successor structure $\mathfrak{A}$ over the alphabet Σ is a first order structure $\mathfrak{A} = (A, \Sigma, +1_{l_1}, \ldots, +1_{l_n})$ where A is a finite set, $+1_{l_1}, \ldots, +1_{l_n}$ are successor relations of n linear orders over A. For notational convenience sometimes we represent a n-SS as $\mathfrak{A} = (A, \lambda, +1_{l_1}, \ldots, +1_{l_n})$. We denote the linear order corresponding to $+1_{l_i}$ by the symbol $\leq_{l_i}$. Restricting the structure $\mathfrak{A}$ to the order $\leq_{l_i}$ yields a word, we call the word $(A, \Sigma, +1_{l_i})$ the word/string projection of $\mathfrak{A}$ to the successor $+1_{l_i}$. Henceforth we abbreviate the term n-successor structure as n-SS .

6.6.1 Successor Types

Given $x \in A$, we define the notion of successor type of x in the following way. First of all, we define two equivalences $s(x)$ and $p(x)$ on the set $[n]$ as follows.

$$i \sim_{s(x)} j \Leftrightarrow \exists y \ \left(x + 1_{l_i} y \wedge x + 1_{l_j} y\right)$$
$$i \sim_{p(x)} j \Leftrightarrow \exists y \ \left(y + 1_{l_i} x \wedge y + 1_{l_j} x\right)$$

Note that the relations $s(x)$ and $p(x)$ are equivalences on the set $[n]$. Let $s(x) = \{\zeta_1, \ldots, \zeta_k\}$ and $p(x) = \{\eta_1, \ldots, \eta_l\}$, $\zeta_i, \eta_i \in \mathcal{P}([n])$ be the equivalence classes of $s(x)$ and $p(x)$. Finally, we define the partial morphism $f(x) \subset s(x) \times p(x)$ in the following way.

$$(\zeta_i, \eta_j) \in f(x) \Leftrightarrow s \in \zeta_i, t \in \eta_j, \exists y \ \ (y + 1_{l_s} x \wedge x + 1_{l_t} y)$$

For every x, we call the triple $\tau(x) = (p(x), s(x), f(x))$ the successor type of x. There are only exponentially many such triples for every n. we denote the set of all successor types by Υ. Given a n-SS $\mathfrak{A} = (A, \lambda, +1_{l_1}, \ldots, +1_{l_n})$ we call $(A, \lambda', +1_{l_i})$ where $\lambda' : A \to \Sigma \times \Upsilon$ defined as $\lambda'(x) = (\lambda(x), \tau(x))$ the annotated string projection to the order i.

6.7 Automata on n-SS

In the following we define the notion of an n-SS automaton. Fix an alphabet Σ. An n-SS automaton $\mathcal{A} = (B_1, \ldots, B_n)$ is a composite automaton consisting of n word automata $B_1, \ldots, B_n$.

For $1 \leq i < n$, the automaton B_i is a non-deterministic letter-to-letter word transducer with an input alphabet $\Sigma_i \times \Upsilon$ and an output alphabet Σ_{i+1}. For the transducer B_1 it is the case that $\Sigma_1 = \Sigma$. The automaton B_n is a finite state recognizer with the alphabet $\Sigma_n \times \Upsilon$.

Definition 6.7.1. *Formally, an n-SS automaton $\mathcal{A} = (B_1, \ldots, B_n)$ is a composite automaton consisting of n word automata $B_1, \ldots, B_n$ where, for $1 \leq i < n$, the word transducer B_i is given by the tuple $B_i = (Q_i, \Sigma_i \times \Upsilon, \Sigma_{i+1}, O_i, \Delta_i, q_i, F_i)$, where Q_i is the finite set of states, $\Sigma_i \times \Upsilon$ is the input alphabet, Σ_{i+1} is the output alphabet, $q_i \in Q_i$ is the initial state, $F_i \subseteq Q_i$ is the set of final states, $\Delta_i \subseteq Q_i \times \Sigma_i \times \Upsilon \times Q_i$ is the set of transitions and $O_i : \Delta_i \to \Sigma_{i+1}$ is the output function.*

Given $\mathfrak{A} = (A, \lambda, +1_{l_1}, \ldots, +1_{l_n})$ the automaton B_i runs over the word $w_i = (A, (\lambda_i, \tau), +1_{l_i})$. For the automaton B_1 we fix $\lambda_1 = \lambda$. We define a run $\rho_i : A \to \Delta_i$ of B_i as a labelling such that:

- $\rho_i(\mathsf{min}(+1_{l_i}))$ is a transition from the state q_i.
- if $\rho_i(a) = (p, (\sigma, \tau), q)$ then $\lambda_i(a) = \sigma$ and $\tau(a) = \tau$.
- if $a + 1_{l_i} b$ and $\rho_i(a) = (p, (\sigma, \tau), q)$ and $\rho_i(b) = (p', (\sigma', \tau'), q')$ then $q = p'$.

The run ρ_i is accepting if $\rho_i(\mathsf{max}(+1_{l_i}))$ is a transition to a state in F_i. An accepting run ρ_i of B_i on w_i uniquely defines an output string $w_{i+1} = (A, \lambda_{i+1}, +1_{l_i})$ where $\lambda_{i+1} : A \to \Sigma_{i+1}$ given by $\lambda_{i+1}(a) = O_i(\rho_i(a))$.

The automaton $B_n = (Q_n, \Sigma_n, \Delta_n, q_n, F_n)$ with the set of states Q_n, the initial state $q_n \in Q_n$, the set of final states $F_n \subseteq Q_n$ and the transition relation $\Delta_n \subseteq Q_n \times \Sigma_n \times Q_n$. We can define a run $\rho_n : A \to \Delta_n$ similarly as above. The run ρ_n is accepting if $\rho_n\left(\mathsf{max}\left(+1_{l_n}\right)\right)$ is a transition to a state in F_n.

Given $\mathfrak{A} = (A, \lambda, +1_{l_1}, \ldots, +1_{l_n})$, We say the n-SS automaton $\mathcal{A}$ has an accepting run $\rho = \rho_1, \ldots \rho_n$ if,

- for $1 \leq i < n$, ρ_i is an accepting run of B_i on $\left(A, \left(\lambda_i, \tau\right), +1_{l_i}\right)$ outputing the word $(A, \lambda_{i+1}, +1_{l_i})$,
- ρ_n is an accepting run of B_n on $(A, \lambda_n, +1_{l_n})$.

Given an n-SS automaton $\mathcal{A}$ over Σ, we define $L(\mathcal{A})$ as the set of n-SS $\mathfrak{A}$ over Σ such that $\mathcal{A}$ has an accepting run on $\mathfrak{A}$. Given a set of n-SS L, we say L is recognizable if there is a n-SS automaton $\mathcal{A}$ such that $L = L(\mathcal{A})$. By $\emptyset$, we denote the empty language of n-SS over Σ. Given L, by $\bar{L}$ we denote the set of all $\mathfrak{A}$ over Σ such that $\mathfrak{A} \notin L$. Similarly, given L_1, L_2, by $L_1 \cup L_2$ (alt. $L_1 \cap L_2$) we denote the set of all $\mathfrak{A}$ over Σ such that $\mathfrak{A} \in L_1$ or (alt. and) $\mathfrak{A} \in L_2$.

The following three lemmas are obvious generalizations of the corresponding lemmas for 2-SS automata.

Lemma 6.7.2. *There exists n-SS automata $\mathcal{A}_\emptyset$ and $\mathcal{A}_{\bar{\emptyset}}$ such that $L\left(\mathcal{A}_\emptyset\right) = \emptyset$ and $L\left(\mathcal{A}_{\bar{\emptyset}}\right) = \bar{\emptyset}$.*

Lemma 6.7.3. *Given a regular language $L \subseteq \Sigma^*$, there is a n-SS automaton $\mathcal{A}$ accepting all n-SS whose string projections to the order $\leq_{l_i}$ is in L.*

Lemma 6.7.4. *Languages recognized by n-SS automata are closed under union, intersection and renaming.*

6.8 Logical Characterization of n-SS Automata

Lemma 6.8.1. *For every n-SS automaton $\mathcal{A}$ there is a formula $\varphi_{\mathcal{A}}$ in the logic* $\mathrm{EMSO}^2\left(\Sigma, +1_{l_1}, \ldots, +1_{l_n}\right)$ *such that $L\left(\mathcal{A}\right) = L\left(\varphi_{\mathcal{A}}\right)$.*

Proof. As usual, the idea is to encode a successful run of $\mathcal{A}$ as a formula $\varphi_{\mathcal{A}}$ in $\mathrm{EMSO}^2(\Sigma, +1_{l_1}, \ldots, +1_{l_n})$. From the classical encoding of automata, we know that for $1 \leq i < n$ the run of each B_i can be coded as a formula $\varphi_i(\Sigma_i, \Upsilon, \Sigma_{i+1}, +1_{l_i})$. In the formula φ_i, the unary predicates Σ_i, Υ and Σ_{i+1} occur as free variables. The automaton B_n is encoded as a formula $\varphi_i(\Sigma_i, \Upsilon, +1_{l_i})$. $\square$

In this section we show that given an $\mathrm{FO}^2(\Sigma, +1_{l_1}, \ldots, +1_{l_n})$ formula we can transform it into an equivalent n-SS automaton. First of all, given a formula $\varphi \in \mathrm{FO}^2(\Sigma, +1_{l_1}, \ldots, +1_{l_n})$ we transform it into an equivalent formula in *Scott Normal Form*, $\exists R_1 \ldots R_n \left(\forall x \forall y\ \chi \wedge \bigwedge_j \forall x \exists y\ \psi_j\right)$, where the predicates R_l are unary, and χ and ψ_j are quantifier-free formulas in $\mathrm{FO}^2(\Sigma, +1_{l_1}, \ldots, +1_{l_n})$. Earlier we observed that n-SS automata are closed under renaming and intersection. Therefore it suffices to show that we can construct a n-SS automaton for each of the formulas $\forall x \forall y\ \chi$ and $\forall x \exists y\ \psi_j$ and this is shown by the following two lemmas.

In the following a *unary type* is a one variable quantifier-free formula containing only unary predicates.

Lemma 6.8.2. *Given an* $\mathrm{FO}^2(\Sigma, +1_{l_1}, \ldots, +1_{l_n})$ *formula of the form* $\varphi = \forall x \forall y\ \chi$ *where* χ *is quantifier free, an equivalent* n*-SS automaton of doubly exponential size can be constructed.*

Proof. What follows is a simple generalization of the proof of Lemma 6.4.1. The chain of arguments is exactly the same.

We start by writing φ in CNF causing an exponential blowup in the size of the formula, followed by distributing the universal quantification over the conjunctions and rewriting the formula as $\bigwedge_j \forall x \forall y\ \chi_j$ where each χ_j is of the form,

$$\chi_j = \alpha(x) \vee \beta(y) \vee \epsilon(x,y) \vee \left(\bigvee_{i \in [n]} \delta_i(x,y)\right).$$

Above $\alpha(x)$ and $\beta(y)$ are unary types. The formulas in the group $\epsilon(x,y)$ are $x = y$ and $x \neq y$. The formula δ_i is a disjunction of literals from the set O_i, where

$$O_i = \{+1_{l_i}(x,y), \neg{+1_{l_i}}(x,y), +1_{l_i}(y,x), \neg + 1_{l_i}(y,x)\}.$$

It is enough to construct an n-SS automaton for each χ_j since the automata are closed under intersection. We note that whenever χ_j describes a regular property, we can construct an equivalent n-SS automaton by converting the finite state automaton equivalent to χ_j. If only one clause $\delta_i(x, y)$ is present in χ_j, the formula χ_j describes a regular property over one linear order, namely the order $\leq_{l_i}$. Therefore we restrict our attention to those χ_j where at least two clauses δ_i and δ_k, $i \neq k$ are present. Suppose $\epsilon(x, y) \equiv x = y \vee x \neq y$. In this case, the formula is tautology hence we construct a n-SS automaton which accepts all n-SS.

The case when $\epsilon(x, y) \equiv x \neq y$, as in the proof of Lemma 6.4.1 the formula describes a regular property.

When $\epsilon(x, y) \equiv x = y$ and $\delta_m(x, y)$ contains a negative literal for some $m \in [n]$, we can rewrite χ_j in the form,

$$\chi_j \equiv (\alpha'(x) \wedge \beta'(y) \wedge x \neq y \wedge \delta'_m(x, y)) \rightarrow \left(\bigvee_{i \in [n], i \neq m} \delta_i(x, y) \right),$$

where $\alpha', \beta', \delta'_m$ are the negations of α, β, δ'_m respectively. It follows that $\delta'_m(x, y)$ contains a positive literal and the automaton can verify the formula χ_j by looking at the annotated string projection to the order $\leq_{l_m}$.

The interesting case is when $\epsilon(x, y) \equiv x = y$, and none of $\delta_1, \ldots, \delta_n$ contains a negative literal, that is when $\delta_1, \ldots, \delta_n$ are disjunctions of positive literals. We rewrite χ_j in the following form,

$$\chi_j = (\alpha'(x) \wedge \beta'(y) \wedge x \neq y) \rightarrow \left(\bigvee_{i \in [n]} \delta_i(x, y) \right).$$

The formula says the following. Whenever α' holds at x and β' holds at y and x, y are distinct then they are neighbours in at least one order $\leq_{l_i}$ as dictated by δ_i. If there is no α' in the word there can be any number of β'. Similarly there can be any number of α' if there is no β' occurring in the word. The automaton can guess both these cases and verify them easily. When there is at least one α' and β' present in the word the number of α' and β' are bounded, since all α' and β' has to be neighbours in at least one of the successor relations as dictated by $\bigvee_{i \in [n]} \delta_i(x, y)$

and there are only bounded number of neighbours (atmost $2n$) for any position. Therefore in this case the formula χ_j can be checked by a n-SS automaton by labelling the α' and β'.

$\square$

Lemma 6.8.3. *For each* $\mathrm{FO}^2(\Sigma, +1_{l_1}, \ldots, +1_{l_n})$ *formula of the form* $\forall x \exists y\ \psi$ *where* ψ *is quantifier free, an equivalent 2-SS automaton of doubly exponential size can be constructed.*

Proof. The argument follows the proof of Lemma 6.4.2 with minor adjustments. Begin by writing ψ as an exponential size conjunction of disjunctions of the form $\forall x \exists y \bigwedge_s \bigvee_t (\alpha_s(x) \to \theta_{st}(x, y))$, where α_s enumerates through all possible maximal types, that is $\bigvee_s(\alpha_s(x))$ is a tautology and $(\alpha_s(x) \wedge \alpha'_s(x))$ is unsatisfiable for all $s \neq s'$. The formula θ_{st} is either $\perp$ or of the form,

$$\beta(y) \wedge \epsilon(x, y) \wedge \left(\bigwedge_{i \in [n]} \delta_i(x, y) \right),$$

where, β is a type, ϵ is one of $x = y$, $x \neq y$, δ_i is in O_i.

The premise occurring in distinct conjuncts are distinct (and mutually exclusive). Hence it is possible to distribute the $\forall x \exists y$ over the conjunction. The resulting formula is of the form,

$$\bigwedge_s \forall x \exists y \bigvee_t (\alpha_s(x) \to \theta_{st}(x, y)).$$

We eliminate the disjunction by adding to every disjunct a new unary predicate $\Lambda_{st}(x)$ (these predicates are chosen to be distinct for each ψ) which denotes that at the position x, the t-th disjunct is witnessing α_s. We can rewrite every conjunct in the above formula as,

$$\exists \Lambda_{s1} \Lambda_{s2} \ldots \Lambda_{sk} \left(\forall x \bigvee_t \Lambda_{st}(x) \right) \wedge \left(\bigwedge_t \forall x \exists y\ \ ((\alpha_s(x) \wedge \Lambda_{st}(x)) \to \theta_{st}(x, y)) \right)$$

An n-SS automaton can guess the predicates Λ_{st}. So it is enough to construct an n-SS automaton for each formula of the form $\forall x \exists y\ \ (\alpha_s(x) \to \theta_{st}(x, y))$. If the consequent is false, the language is regular. So we concentrate on the cases where the consequent is satisfiable.

$$\forall x \exists y \left(\alpha(x) \rightarrow \left(\beta(y) \wedge \epsilon(x,y) \wedge \left(\bigwedge_{i \in [n]} \delta_i(x,y) \right) \right) \right)$$

We do a case analysis. If $\epsilon(x,y) \equiv x = y$, the language is regular. Hence now onwards we fix ϵ to be $x \neq y$.

As in the previous proof, we have two cases, when δ_m contains a positive literal for some $m \in [n]$ and when none of $\delta_1, \ldots, \delta_n$ contains a positive literal. If δ_m contains a positive literal for some $m \in [n]$, we can easily verify the formula by looking at the annotated string projection to the appropriate order.

The only remaining case is when none of $\delta_1, \ldots, \delta_n$ contains a positive literal. In this case, we claim that there exists a $k \in \mathbb{N}$ such that if there are at least k many β in A then it is guaranteed that every α has a witness. Therefore the automaton guesses one of the following,

- the number of β present in A is fewer than k. In this case the automaton guesses the number of β and verifies that. In addition, it labels each β with a distinct label and verifies that for every α there is at least one β witnessing it.
- the number of β present in A is at least k. In this case the automaton just verifies that the guess is correct.

$\square$

This gives us that,

Theorem 6.8.4. *A n-SS language is recognizable if and only if it is definable in* $\mathrm{EMSO}^2(\Sigma, +1_{l_1}, \ldots, +1_{l_n})$.

6.8.1 Discussion

We saw that there is a natural generalization 2-SS automaton to n successor structures. The proof of equivalence between 2-SS and $\mathrm{EMSO}^2(\Sigma, +1_{l_1}, +1_{l_2})$ extends seamlessly in this case. However the emptiness problem for n-SS automaton remains open. The proof for the two successor case fails to generalize because of the presence of more than one markings.

7

Ordered data words

7.1 Introduction

In this chapter we study two variable logic and data automata on ordered data words. We model ordered data words as first-order structures with a linear order and a total preorder (as discussed in Chapter 5). Henceforth we refer to the total preorder as preorder and totality is assumed. The contents of this chapter is part of a joint work with Thomas Zeume [MZ11].

In the following we focus our attention to the finite satisfiability problem of $\mathrm{FO}^2(\Sigma, +1_l, \leq_p, +1_p)$ (See Chapter 5 for the context). However we do not solve the general case here. Instead, we show that finite satisfiability of $\mathrm{FO}^2(\Sigma, +1_l, \leq_p, +1_p)$ is decidable over $(+1_l, \leq_p, +1_p)$-structures where each equivalence class of $\leq_p$ contains at most k elements for a fixed k. We will shortly describe the rationale behind this restriction. Since 1-boundedness can be axiomatized in FO^2 by $\forall x \forall y (x \neq y \rightarrow \neg x \sim_p y)$, the following is an immediate corollary: The finite satisfiability problem for $\mathrm{FO}^2(+1_{l_1}, \leq_{l_1}, +1_{l_2})$ is decidable.

It is known that the finite satisfiability problem of $\mathrm{FO}^2(\Sigma, \leq_l, +1_l, \sim_p)$ and non-emptiness problem of data automata over 2-bounded data words (data words where class has length 2) are as hard as reachability in vector addition systems. Not only this, the data languages described in Chapter 3, fall into this class. This is sufficient evidence to infer that we are dealing with a non-trivial subclass of data words. In the latter part of this chapter, we will see that in fact the finite

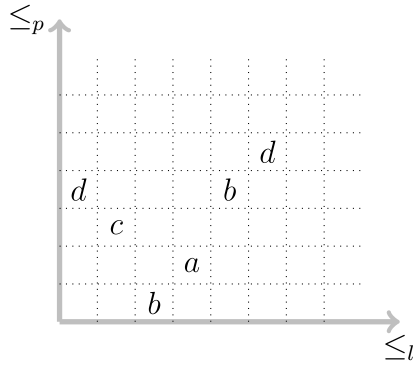

Figure 7.1: A $(+1_l, +1_p, \leq_p)$-structure and representation in the plane. Columns are ordered by $\leq_l$, rows are ordered $\leq_p$

satisfiability problem for $\mathrm{FO}^2(\Sigma, +1_l, \leq_p, +1_p)$ over k-bounded ordered data words is as hard as reachability in vector addition systems.

7.2 Automata over ordered data words

We observe that ordered data words (words with an additional total preorder on their positions) – have a natural representation as sets of labeled points in the two-dimensional plane, see Figure 7.1 for the representation of the ordered data word w,

$$w = \begin{pmatrix} d & c & b & a & b & d \\ 4 & 3 & 1 & 2 & 4 & 5 \end{pmatrix}$$

We will use this intuition in the following constructions and proofs.

In the following, we introduce some vocabulary for $(+1_l; +1_p, \leq_p)$-structures. These structures are also called *ordered data words*. An ordered data word is k-bounded if $\leq_p$ is k-bounded. We call such structures k-ordered data words (abbreviated as k-o.d.w). Let $\mathfrak{A}$ be a k-bounded $(+1_l; +1_p, \leq_p)$-structure over universe A. The *string projection ($\leq_l$-projection)* $sp(\mathfrak{A})$ of $\mathfrak{A}$ is the restriction of $\mathfrak{A}$ to unary relations as well as the $+1_l$-relation, i.e. $sp(\mathfrak{A}) = (A, \Sigma, +1_l)$. The $\leq_l$-projection is identified with the sequence of the labels of all elements in linear order. For example, the $\leq_l$-projection of the $(+1_l; +1_p, \leq_p)$-structure from Figure

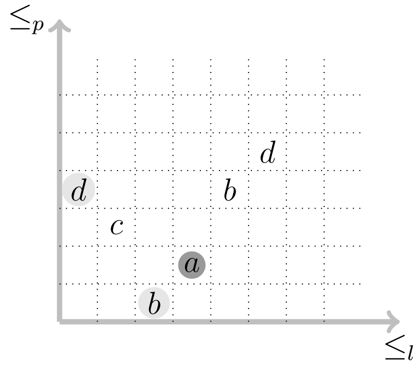

Figure 7.2: A $(+1_l, +1_p, \leq_p)$-structure and markings. Columns are ordered by $\leq_l$, rows are ordered $\leq_p$, i.e. every box represents the intersection of one $\leq_p$-class and one $\leq_l$-class. The markings of the a, b, d are respectively $+\infty, +1, -1$.

7.1 is d, c, b, a, b, d.

For alphabet Σ, define $parikh(\Sigma) = \mathbb{N}^{|\Sigma|}$.

Slightly abusing the notation we use $+1_p$ and $\leq_p$ to denote the successor relation and linear order relation on $A/{\sim_p}$ that are induced by $\leq_p$. Let $parikh_k(\Sigma)$ be the set of parikh vectors over Σ whose sum of components is at most k. Every $[a] \in A/{\sim_p}$ can be labeled by a symbol p from $parikh_k(\Sigma)$ such that, for every $\sigma \in \Sigma$, p indicates the number of σ-labeled elements in $[a]$. The *preorder projection ($\leq_p$-projection)* of $\mathfrak{A}$ is $pp(\mathfrak{A}) = (A/{\sim_p}, parikh_k(\Sigma), +1_p, \leq_p)$, i.e. the $\leq_p$-projection of $\mathfrak{A}$ considers $\sim_p$-equivalence classes as single elements. We will identify the preorder projection with the sequence $p_1, \ldots, p_m$ where $p_i \in parikh_k(\Sigma)$ is the parikh image of the ith element of $A/{\sim_p}$ with respect to $\leq_p$. Thus the preorder projection of the $(+1_l; +1_p, \leq_p)$-structure from Figure 7.2 is $(0,1,0,0), (1,0,0,0), (0,0,1,0), (0,1,0,1), (0,0,0,1)$ where, for instance, the second component of all vectors represents the number of occurrences of the label b.

Recall that we write $+1^s_p(u, v)$ if the equivalence class of v with respect to $\leq_p$ is the successor of the equivalence class of u. Further we say u and v are $+1_p$-close, if either $u{+}1^s_p v$ or $u \sim_p v$ or $v + 1^s_p u$. If $u \leq_p v$ and if they are not $+1_p$-close, we denote it by $u \ll_p v$.

Let Γ_l be the alphabet $\{-\infty, -1, 0, 1, +\infty\}$. As before, we annotate the string projection with a marking as follows. Given $a \in A$ we define the marking of a on $+1_l$, $M_l(a) \in \Gamma_l$ as,

$$M_l(a) = \begin{cases} -\infty & \textit{if} \;\; \exists b \; (a + 1_l b \wedge b \ll_p a) \\ -1 & \textit{if} \;\; \exists b \; (a + 1_l b \wedge b + 1_p^s a) \\ 0 & \textit{if} \;\; \exists b \; (a + 1_l b \wedge a \sim b) \\ +1 & \textit{if} \;\; \exists b \; (a + 1_l b \wedge a + 1_p^s b) \\ +\infty & \textit{if} \;\; \exists b \; (a + 1_l b \wedge a \ll_p b) \vee \neg \exists b \;\; (a + 1_l b) \end{cases}$$

The above marking we call the $\leq_p$-marking as it encodes the distance between two consecutive positions of the linear order with respect to the preorder. Given $\mathfrak{A} = (A, \Sigma, +1_l, \leq_p, +1_p)$, we define the marked string projection of $\mathfrak{A}$ as the word $(A, \Sigma, \Gamma_l, +1_l)$ where each position is annotated with its marking.

7.3 k-bounded Ordered Data Automaton

We propose a variant of Data automata on k-class bounded ordered data words. Register automata have been extended to the case of ordered data words recently [ST11]. However, the automata presented below are incomparable with the mentioned generalization.

Definition 7.3.1. *A k-bounded Ordered Data Automaton (k-ODA) is a composite automaton $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ where $\mathcal{B}$ is a non-deterministic finite state transducer with an input alphabet $\Sigma \times \Gamma_l$ and an output alphabet Π. The automaton $\mathcal{C}$ is a finite state automaton working on words over the alphabet parikh$_k(\Pi)$.*

The set-up of k-ODA is very similar to that of 2-SS automaton seen in the last chapter, except that instead of a linear successor relation $+1_{l_2}$, we have a total preorder. Intuitively the transducer $\mathcal{B}$ is running over $\mathfrak{A}$ with respect to the linear order induced by $+1_l$. The automaton $\mathcal{C}$ runs on the result of $\mathcal{B}$ with respect to $\leq_p$.

Given a $(+1_l, +1_p, \leq_p)$-structure $\mathfrak{A}$ a k-ODA $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ works on $\mathfrak{A}$ as follows. The transducer $\mathcal{B}$ works on the marked string projection of $\mathfrak{A}$ yielding a run ρ_B which in turn defines the unique (for each run) new structure $\mathfrak{A}'$. The automaton $\mathcal{C}$

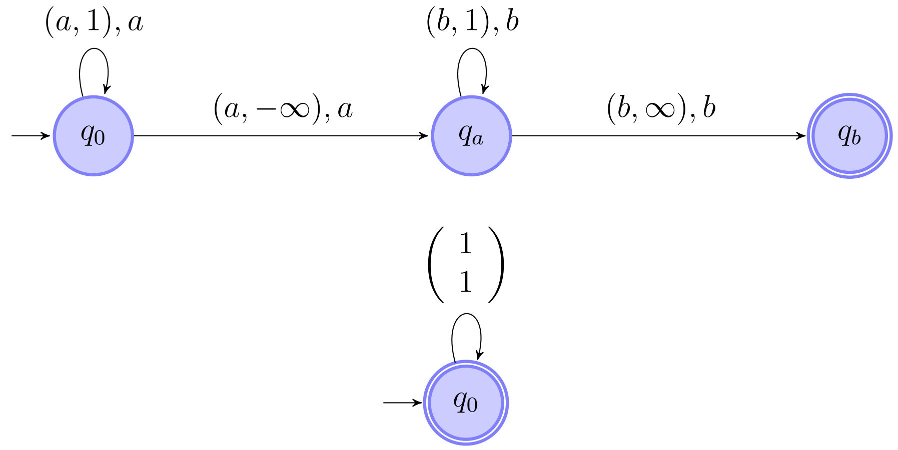

Figure 7.3: 2-ODA accepting L_{ww}

runs over the preorder projection of the structure $\mathfrak{A}'$ yielding a run ρ_C. Automaton $\mathcal{A}$ accepts the structure $\mathfrak{A}$ if both ρ_B and ρ_C are accepting. The set of all k-bounded structures accepted by $\mathcal{A}$ is denoted by $L(\mathcal{A})$. The transducer B is called *linear order automaton*, the automaton C is called *preorder automaton*.

Example 7.3.2. *Let $\mathcal{A} = (B, C)$ the 2-ODA over the alphabet $\Sigma = \{a, b\}$ shown in Figure 7.3. The transducer B is a copy machine and accepts the language $(a,1)^*(a,-\infty)(b,1)^*(b,\infty)$. This ensures that any two identically labelled positions u and v, if they are consecutive in the linear order then they are consecutive in the preorder as well. The preorder automaton accepts the language $(1,1)^*$, that is C specifies that all equivalence classes of $\leq_p$ contain exactly an 'a' and a 'b'.*

Let w be the following 2-o.d.w.

$$w = \begin{matrix} a & a & a & a & a & a & b & b & b & b & b & b \\ 1 & 2 & 3 & 4 & 5 & 6 & 1 & 2 & 3 & 4 & 5 & 6 \end{matrix}$$

The marked string projection of w is,

$$sp(w) = \begin{matrix} a & a & a & a & a & a & b & b & b & b & b & b \\ 1 & 1 & 1 & 1 & 1 & -\infty & 1 & 1 & 1 & 1 & 1 & \infty \end{matrix}$$

The preorder projection of w is,

$$pp(w) = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Transducer B has a successful run on $sp(w)$ and the automaton C has an accepting run on $pp(w)$. Hence w is accepted by the automaton $\mathcal{A}$. Let L_{ww} be the language accepted by the automaton $\mathcal{A}$. We claim that,

$$L_{ww} = \{(a, d_1) \dots (a, d_n)(b, d_1) \dots (b, d_n) \mid n \geq 3, \forall i \leq n : \ +1_p\,(i, i+1)\}$$

Observe that B specifies that the data values under a as well as b are strictly increasing and all a's are preceded by b's in the linear order. Since C ensures that all equivalence classes contain exactly an 'a' and a 'b', it is easy to see that number of a's in w is same as number of b's. These two facts together imply our claim.

Example 7.3.3. *Consider the automaton $\mathcal{A} = (B, C)$ the 2-ODA over the alphabet $\Sigma = \{a, b\}$ shown in Figure 7.4. The transducer B is a copy machine and accepts the language $(a, 1)^*(a, 0)(b, -1)^*(b, \infty)$. This ensures two a-labelled positions u and v, if they are consecutive in the linear order then they are consecutive in the preorder as well. Similarly, if u and v are labelled by b and $+1_l(u, v)$ then $+1_p(v, u)$. As in the previous example the preorder automaton accepts the language $(1, 1)^*$, that is C specifies that all equivalence classes of $\leq_p$ contain exactly an 'a' and a 'b'.*

Let w be the following 2-o.d.w.

$$w = \begin{array}{cccccccccccc} a & a & a & a & a & a & b & b & b & b & b & b \\ 1 & 2 & 3 & 4 & 5 & 6 & 6 & 5 & 4 & 3 & 2 & 1 \end{array}$$

$$sp(w) = \begin{array}{cccccccccccc} a & a & a & a & a & a & b & b & b & b & b & b \\ 1 & 1 & 1 & 1 & 1 & 0 & -1 & -1 & -1 & -1 & -1 & \infty \end{array}$$

The preorder projection of w is,

$$pp(w) = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

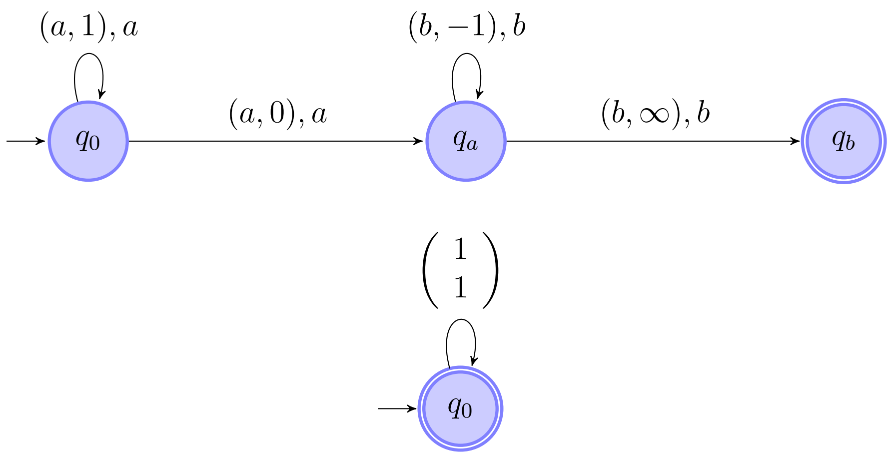

Figure 7.4: 2-ODA accepting L_{ww^r}

The automaton B and C, both have a successful run on $sp(w)$ and $pp(w)$ respectively. Hence w is accepted by the automaton $\mathcal{A}$. Let L_{ww^r} be the language accepted by the automaton $\mathcal{A}$. We claim that,

$$
\begin{aligned}
L_{ww^r} = \{(a,d_1)\ldots(a,d_n)(b,d_n)\ldots(b,d_1) \mid n \geq 2, \quad & \forall 1 \leq i \leq n: \ +1_p\,(i,i+1)\,, \\
& \forall n \leq i \leq 2n: \ +1_p\,(i+1,i)\}
\end{aligned}
$$

The transducer B ensures that the data values under a are strictly increasing and that the data values under b are strictly decreasing and all a's are preceded by b's in the linear order. Since C ensures that all equivalence classes contain exactly an 'a' and a 'b', it is easy to observe that number of a's in w is same as number of b's. These two facts together imply our claim.

Firstly, we examine some properties of k-ODA. The following lemmata can be proved straightforwardly as in the case of 2-SS automata.

Lemma 7.3.4. *There exist k-ODA $\mathcal{A}_\emptyset$ and $\mathcal{A}_{\bar{\emptyset}}$ which accept no k-bounded $(+1_l, +1_p, \leq_p)$-structure and all k-bounded $(+1_l, +1_p, \leq_p)$-structures, respectively.*

Lemma 7.3.5. *Languages accepted by k-ODA are closed under union, intersection and renaming.*

Proof. Closure under union and intersection are proved by using the usual product construction, closure under renaming by using the non-determinism of $\mathcal{B}$. $\square$

The following proposition can be proved like Lemma 6.3.8 from Chapter 5.

Proposition 7.3.6. *Languages accepted by k-ODA are not closed under complementation.*

Given a formula $\varphi \in \mathrm{EMSO}^2(\Sigma, +1_l, +1_p, \leq_p)$ we define $L_k(\varphi)$ as the set of all k-o.d.w. w such that $w \models \varphi$.

Proposition 7.3.7. *Given a k-ODA $\mathcal{A}$ there is a formula $\varphi_{\mathcal{A}} \in \mathrm{EMSO}^2(\Sigma, +1_l, +1_p, \leq_p)$ such that $L(\mathcal{A}) = L_k(\varphi_{\mathcal{A}})$.*

Proof. We know that there is a formula φ_B in $\mathrm{EMSO}^2(\Sigma, \Gamma_l, \Pi, +1_l)$ which encodes a successful run of B. It is easy to write down a formula in φ_{Γ_l} in $\mathrm{FO}^2(\Gamma_l, +1_l, +1_p, \leq_p)$ which verify the validity of the Γ_l predicates.

Assume that the input structure is labelled by unary predicates from Π according to φ_B. We write a formula in φ_C in $\mathrm{EMSO}^2(\Pi, +1_p)$ which encodes a successful run of C. Let $C = (Q, \Pi, \Delta, I, F)$. Using unary predicates X_{δ_i} for each $\delta_i \in \Delta$ and Y_i for each $0 \leq i \leq k$ we write down a formula φ_C encoding a successful run of C over the structure in the following way.

$$\varphi_C = \exists X_{\delta_1} \dots X_{\delta_n} \exists Y_1 \dots Y_k \left(\varphi_X \wedge \varphi_Y \wedge \varphi_\sim \wedge \bigwedge_{\delta \in \Delta} \varphi_\delta \wedge \varphi_\Delta \wedge \varphi_I \wedge \varphi_F \right)$$

where the individual formulas verify the following;

$$\varphi_X = \forall x \bigvee_{\delta_i \in \Delta} X_{\delta_i}(x) \wedge \forall x \bigwedge_{\delta_i \neq \delta_j} \neg \left(X_{\delta_i}(x) \wedge X_{\delta_j}(x) \right)$$

φ_X says that every element is labelled by exactly one X predicate.

$$\varphi_Y = \forall x \bigwedge_{i \neq j} \neg \left(Y_i(x) \wedge Y_j(x) \right) \wedge \forall x \forall y \left(x \sim_p y \wedge P_a(x) \wedge P_a(y) \rightarrow \bigwedge_i \neg \left(Y_i(x) \wedge Y_i(y) \right) \right)$$

φ_Y says that all elements in the same equivalence class labelled by the same Π predicate have distinct Y labels.

$$\varphi_\sim = \forall xy \bigwedge_{\delta_i \in \Delta} (x \sim y \wedge X_{\delta_i}(x) \rightarrow X_{\delta_i}(y))$$

$\varphi_\sim$ states that all elements in the same class has precisely the same X_δ label.

Let $\delta = (p, \bar{c}, q)$ where $\bar{c} = (c_1, \ldots, c_n) \in \mathit{parikh}_k(\Pi)$. The formula φ_δ verifies that if an element is labelled by X_δ then the class satisfies the constraint. This is achieved by counting the number of Y predicates in each class.

$$\varphi_\delta = \forall x \left(X_\delta(x) \rightarrow \bigwedge_{a_i \in \Pi} \varphi_{a_i}^{c_i}(x) \right)$$

where;

$$\varphi_{a_i}^{c_i}(x) = \forall y \left(x \sim y \wedge P_{a_i}(x) \rightarrow \bigvee_{0 \leq j \leq c_i} Y_j \right) \wedge \left(\bigwedge_{0 \leq j \leq c_i} (\exists y \; P_a(y) \wedge Y_j(y) \wedge x \sim y) \right)$$

Finally we ensure that the automaton has a valid run. It starts in the initial state (ensured by formula φ_I), ends in a final state (ensured by formula φ_F) and every two consecutive transitions share a common state (ensured by φ_Δ).

$$\varphi_\Delta = \forall xy \left(+1_p(x,y) \wedge \neg{+1_p}(y,x) \rightarrow \bigvee_{\mathsf{end}(\delta_i) = \mathsf{start}(\delta_j)} \big(X_{\delta_i}(x) \wedge X_{\delta_j}(y) \big) \right)$$

$$\varphi_I = \forall x \left(\neg \exists y \, (+1_p(y,x) \wedge \neg{+1_p}(x,y)) \rightarrow \bigvee_{\delta \in \Delta_{init}} X_\delta(x) \right)$$

$$\varphi_F = \forall x \left(\neg \exists y \, (+1_p(x,y) \wedge \neg{+1_p}(y,x)) \rightarrow \bigvee_{\delta \in \Delta_{final}} X_\delta(x) \right)$$

Finally, (after pulling out the second order quantifiers) our desired formula expressing the run of $\mathcal{A}$ is,

$$\varphi_A = \exists \Gamma_l \ldots \exists \Pi \ldots \exists Y \ldots \exists X_\delta \ldots (\varphi_{\Gamma_l} \wedge \varphi_B \wedge \varphi_C).$$

$\square$

Next we want to show that given a formula $\varphi \in \mathrm{EMSO}^2(\Sigma, +1_l, +1_p, \leq_p)$ there is an automaton A_φ such that $L(A_\varphi) = L_k(\varphi)$. The logics $\mathrm{EMSO}^2(\Sigma, +1_l, +1_p, \leq_p)$ and $\mathrm{EMSO}^2\left(\Sigma, +1_l, \sim_p, +1_p^s, \ll_p\right)$ are identical in expressive power since the sets of predicates $\{+1_p, \leq_p\}$ and $\{\sim_p, +1_p^s, \ll_p\}$ are mutually definable as shown below.

$$x \sim_p y := +1_p(x,y) \wedge +1_p(y,x)$$

$$+1_p^s(x,y) := +1_p(x,y) \wedge \neg{+1_p}(y,x)$$

$$x \ll_p y := x \leq_p y \wedge \neg{+1_p}(x,y)$$

$$+1_p(x,y) := +1_p^s(x,y) \vee x \sim_p y$$

$$x \leq_p y := x \ll_p y \vee +1_p^s(x,y) \vee x \sim_p y$$

We will be using the logic $\mathrm{EMSO}^2\left(\Sigma, +1_l, \sim_p, +1_p^s, \ll_p\right)$ for convenience. Given a formula φ in $\mathrm{EMSO}^2\left(\Sigma, +1_l, \sim_p, +1_p^s, \ll_p\right)$ we proceed by writing it in Scott Normal Form as

$$\exists X_1 \ldots X_n \left(\forall x \forall y\ \psi \wedge \bigwedge_i \forall x \exists y\ \chi_i \right)$$

where ψ and χ_i are quantifier-free formulas. Since k-ODA are closed under union, intersection and renaming it is sufficient to show that there exist k-ODA accepting models of formulas of the form $\forall x \forall y\ \psi$ and $\forall x \exists y\ \chi$.

Lemma 7.3.8. *Given a formula of the form $\forall x \forall y\ \psi$ there is a k-ODA accepting its k-class bounded models.*

Proof. We first write ψ in CNF and distribute the universal quantifier over the conjunction. This allows us to restrict our attention to formulas of the form (because of closure under intersection)

$$\varphi = \forall x \forall y \left(\alpha(x) \vee \beta(y) \vee \gamma_{=}(x,y) \vee \delta_l(x,y) \vee \delta_p(x,y) \right),$$

where α, β are unary types and the rest of the formulas are as follows. If S is a set of formulas we denote by $\mathsf{Disj}(S)$ the set of disjunctive formulas over S. The formulas $\gamma_{=}(x,y) \in \mathsf{Disj}\,(\Gamma_{=})$, $\delta_l(x,y) \in \mathsf{Disj}\,(\Delta_l)$, $\delta_p(x,y) \in \mathsf{Disj}\,(\Gamma_{\sim} \cup \Delta_p \cup \Theta_p)$ where,

$$\begin{array}{rcl}
\Gamma_{=} & = & \{x = y, x \neq y\} \\
\Gamma_{\sim} & = & \{x \sim_p y, x \not\sim_p y\} \\
\Delta_l & = & \{+1_l(x,y), \neg{+1_l}(x,y), +1_l(y,x), \neg{+1_l}(y,x)\} \\
\Delta_p & = & \{+1^s_p(x,y), \neg{+1^s_p}(x,y), +1^s_p(y,x), \neg{+1^s_p}(y,x)\} \\
\Theta_p & = & \{x \ll_p y, \neg(x \ll_p y), y \ll_p x, \neg(y \ll_p x)\}
\end{array}$$

We can further rewrite $\delta_p(x,y)$ upto logical equivalence as a disjunction of two variable order types on $\leq_p$, that is $\delta_p(x,y) \in \mathsf{Disj}(O_p)$ where

$$O_p = \{x \sim_p y, +1^s_p(x,y), +1^s_p(y,x), x \ll_p y, y \ll_p x\}.$$

What follows is a case analysis. If φ is a tautology then by Lemma 7.3.4 there is an automaton accepting $L(\varphi)$. Hence, without loss of generality assume that φ is not a tautology and that each of $\alpha, \beta, \gamma_{=}, \delta_l, \delta_p$ are not null.

If $\gamma_{=}$ is $x \neq y$ then we can write φ as

$$\varphi = \forall x \forall y \left(\alpha'(x) \wedge \beta'(y) \wedge x = y \rightarrow \delta_l(x,y) \vee \delta_p(x,y) \right).$$

Substituting $x = y$ in the consequent reduces it to True or False. The reduced formula can be checked by a k-ODA. Henceforth we fix $\gamma_{=}$ to be $x = y$.

If δ_l contains a negative formula we can rewrite φ as,

$$\varphi = \forall x \forall y \left(\alpha'(x) \wedge \beta'(y) \wedge x \neq y \wedge \delta'_l(x,y) \rightarrow \delta_p(x,y) \right),$$

where δ'_l is a positive formula upto logical equivalence. It can be seen immediately that φ can be checked by a k-ODA by looking at the marked string projection. Henceforth we assume that δ_l is a positive formula.

To handle this case we bring the δ_p formula to the left of the implication as

$$\varphi = \forall x \forall y \left(\alpha'(x) \wedge \beta'(y) \wedge x \neq y \wedge \delta'_p(x,y) \rightarrow \delta_l(x,y)\right),$$

where δ'_p is a conjunction of negative literals. By virtue of the fact that O_p is a complete set of order types in two variables, we can rewrite δ'_p as a disjunction of positive formulas. Again using the fact that order types are mutually exclusive, the above sentence is logically equivalent to a conjunction of sentences of the form,

$$\varphi = \forall x \forall y \left(\alpha'(x) \wedge \beta'(y) \wedge x \neq y \wedge o_p(x,y) \rightarrow \delta_l(x,y)\right),$$

where $o_p(x,y) \in O_p(x,y)$ or is False. Next we show how to construct a k-ODA for each of these sentences.

When $o_p(x,y)$ is False : In this case the sentence is always true. Hence we construct a k-ODA accepting all k-o.d.w.

When $o_p(x,y) \in \{+1^s_p(x,y), +1^s_p(y,x), x \sim_p y\}$: To verify this sentence we use the following scheme. Let $C = (\{s,t\} \times [k]) \cup (\{s',t'\} \times [k])$ be a set of colours. Whenever $+1_l(x,y)$ and $M_l(x) \neq \pm\infty$ the transducer B labels position x and y in the following way and output. (1) If $M_l(x) = 1$, x is labelled by some (s,i) and y is labelled by (t,i). (2) If $M_l(x) = -1$, x is labelled by some (t,i) and y is labelled by (s,i). (3) If $M_l(x) = 0$, x is labelled by some (s',i) and y is labelled by (t',i). The automaton C verifies that in every class the labels appear uniquely (there may be positions with more than one label). In this way, automaton C is able to make out the neighbourhood relationship in the linear order between positions appearing in the same class or in adjacent classes. Thus the automaton C can easily verify the formula.

When $o_p(x,y) \in \{x \ll_p y, y \ll_p x\}$: All sentences of this form are verified in the similar fashion. We describe only one case. Consider when $o_p(x,y)$ is $x \ll_p y$. We observe that there are only boundedly many (say $l, l \leq k$) β' occurring after the first occurrence of α' (if there is an α' in the structure) in the order $\leq_p$. The automaton verifies the sentence in the following fashion. If α' or β' does not occur in the structure the automaton guesses this and verifies it. Otherwise the B automaton guesses the β' which occur before the first α' in the preorder projection and labels them by $\bar{\beta}$. For the other β', of which there are boundedly many, the automaton assigns unique labels $\beta_1, \ldots, \beta_l, l \leq k$. It also records the neighbourhood relationship in the linear order of each α' with $\beta_1, \ldots, \beta_l, l \leq k$ using some label. The C automaton verifies that all β' occur before any of the α'. It also verifies that every α' satisfies the neighbourhood relationship given by δ_l with those β_i which follows it (strictly) in the preorder projection.

□

Lemma 7.3.9. *Given a formula of the form $\forall x \exists y\ \chi$ there is a k-ODA accepting its k-class bounded models.*

Proof. We begin by writing χ as an exponential-sized conjunction of disjunctions of the form $\forall x \exists y \wedge_i \vee_j (\alpha_i(x) \rightarrow \theta_{ij})$ where α_i are maximal types, that is to say $\vee_i \alpha_i$ is a tautology and $\alpha_i \wedge \alpha_j$ is a contradiction for $i \neq j$. This is achieved using the truth table for χ. The formula θ_{ij} is either False or of the form

$$\beta(y) \wedge \gamma_=(x,y) \wedge \delta_l(x,y) \wedge \delta_p(x,y)$$

where β is a type, $\gamma_=(x,y) \in \Gamma_=$, $\delta_l \in \mathsf{Conj}\,(\Delta_l)$ and $\delta_p \in \mathsf{Conj}\,(\Gamma_p \cup \Delta_p \cup \Theta_p)$. In the previous sentence, $\mathsf{Conj}(S)$ denotes a conjunction of formulas from the set S. By virtue of the fact that O_p is a set of complete and mutually exclusive set of order types in two variables we can rewrite δ_p as a disjunction of literals from O_p and rewrite the formula as $\forall x \exists y \wedge_i \vee_{j'} (\alpha_i(x) \rightarrow \theta_{ij'})$ where $\theta_{ij'}$ is either False or of the form

$$\beta(y) \wedge \gamma_=(x,y) \wedge \delta_l(x,y) \wedge o_p(x,y)$$

where $o_p \in O_p$.

Since α_i are maximal we can pull out the outer conjunction rewriting χ as $\wedge_i \forall x \exists y \vee_{j'} (\alpha_i(x) \to \theta_{ij'})$. We can eliminate the disjunction by adding a new unary predicate $\Lambda_{ij'}$ for each $\theta_{ij'}$ expressing the fact that for the premise α_i then disjunct $\theta_{ij'}$ is chosen as the witness. Every conjunct $\forall x \exists y \vee_{j'} (\alpha_i(x) \to \theta_{ij'})$ is rewritten as

$$\exists \Lambda_{i1} \ldots \exists \Lambda_{ij'} (\forall x \vee_{j'} \Lambda_{ij'}(x)) \wedge \bigwedge_{j'} \forall x \exists y \left[(\alpha_i(x) \wedge \Lambda_{j'}(x)) \to \theta_{ij'}(x, y) \right].$$

The automaton can guess and verify the Λ predicates, so it is enough to construct a k-ODA for each formula of the form $\varphi = \forall x \exists y \, (\alpha(x) \to \theta_{ij'}(x, y))$. If $\theta_{ij'}$ is False, then φ is regular. When $\gamma_=(x, y) \equiv x = y$ then also φ is regular. Henceforth we fix $\gamma_=(x, y) \equiv \neg x = y$.

Whenever $\delta_l(x, y)$ contains a positive literal, the sentence φ can be verified by looking at the marked string projection. Henceforth we fix δ_l to be a negative formula. Next we show how to construct a k-ODA for each of these sentences.

When $o_p(x, y) \in \{+1^s_p(x, y), +1^s_p(y, x), x \sim_p y\}$: In this case we use the scheme used in the last proof to verify the sentence.

When $o_p(x, y) \in \{x \ll_p y, y \ll_p x\}$: The cases are analogous. We treat only the case when $x \ll_p y$. We observe that there is a bound k such that if there are at least k many β' following an α' in the preorder projection then that α' can be witnessed always, irrespective of the linear order. We call those α' with at least k many β' following them in the preorder projection *good* α'. The remaining α' are called *bad* α'.

In two cases the sentence is satisfied trivially, when α' is absent in the structure and when there is no bad α'. Both these cases can be guessed and verified easily. So assume that there is a bad α' occurring in the structure.

This case is verified in the following way. First of all the B automaton guesses the good α' and bad α' by means of some labelling. Let α'_1 be the first occurrence of a bad α' in the preorder projection. All α' following α'_1 in the preorder are also bad α'. The B automaton guesses the β' which follows the α'_1 in the preorder and labels them by $\beta_1, \ldots, \beta_m$. It also guesses the bad α' and labels them by their neighbourhood relationship to $\beta_1, \ldots, \beta_m$ in the linear order. The C automaton verifies the following; (1) The guesses made by the B automaton are correct. (2) Every bad α' has some witness which satisfies the neighbourhood relationship given by δ_l. This completes the proof.

□

Proposition 7.3.7, Lemma 7.3.8 and Lemma 7.3.9 immediately give the following proposition, which in turn shows that satisfiability of $\mathrm{FO}^2\left(\Sigma, +1_l, +1_p, \leq_p\right)$ over k-o.d.w reduces to the nonemptiness problem of k-ODA.

Proposition 7.3.10. *A language L of k-o.d.w is accepted by a k-ODA if and only if there is a formula $\varphi \in \mathrm{EMSO}^2\left(\Sigma, +1_l, +1_p, \leq_p\right)$ such that $L = L_k\left(\varphi\right)$.*

7.3.1 Deciding the Emptiness Problem for k-ODA

Now, we prove the decidability of the emptiness problem of k-ODA by a reduction to the emptiness problem of multicounter automata. Roughly speaking, the run of a k-ODA $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ on an input structure $\mathfrak{A}$ will be simulated by a multicounter automaton that reads the preorder projection of $\mathfrak{A}$, guesses and verifies a run of $\mathcal{C}$, and, meanwhile, builds up a run of $\mathcal{B}$ in the counters. The intricate part of reconstructing a run of $\mathcal{B}$ is that $\mathcal{B}$ sees the $\leq_p$-marking. The proof proceeds as follows. Lemma 7.3.12 deals with those parts of runs of $\mathcal{B}$ that process consecutive positions with $\leq_p$-marking $+1$, 0 and -1, called *block* later. Constructing a complete run of $\mathcal{B}$ from different blocks will mainly be done in Proposition 7.3.13 and Theorem 7.3.15. The former prepares for the latter by introducing some techniques, and solving the 1-bounded case. Theorem 7.3.15 then solves the general case.

A *block* B is a maximal sequence $u_1, \ldots, u_n$ of elements from A such that, for all i in $\{1, \ldots, n-1\}$, u_i and u_{i+1} are close with respect to both the linear order and the preorder. The elements u_1 and u_n are called *leftmost* and *rightmost* positions of B. We denote the leftmost and rightmost positions by $L(B)$ and $R(B)$. Note that the elements u_0 and u_{n+1} with $+1_l(u_0, u_1)$ and $+1_l(u_n, u_{n+1})$ (in case they exist) are not $\leq_p$-close to u_1 and u_n, respectively. If $u_1 \leq_p u_0$ we sometimes write $L^+(B)$ instead of $L(B)$. Similarly for $L^-(B), R^+(B)$ and $R^-(B)$. From now on we will assume that u_1 and u_n are labelled by the appropriate L^+ or L^- and R^+ or R^-. The preorder projection $pp(B)$ of a block B is the preorder projection of $\mathfrak{A}$ restricted to B. Likewise for the linear order projection. As before, we identify the preorder projection $[p_1] \leq_p \ldots \leq_p [p_m]$ of a block with the sequence $\mathit{parikh}([p_1]), \ldots, \mathit{parikh}([p_m])$ over $\mathit{parikh}_k(\Sigma)$. We observe that the image of a block B in the linear order projection and in the preorder projection of $\mathfrak{A}$ is a contiguous interval.

Example 7.3.11. *Let $\mathfrak{A} = (A, +1_l, \leq_p, +1_p)$ be the following k-o.d.w (we avoid the unary labelling),*

$$A = \{a_1, \ldots, a_{18}\}$$

$$+1_l = \{(a_i, a_{i+1}) \mid 1 \leq i \leq 18\}$$

The equivalence classes of $\leq_p$ are ordered as,

$$\{a_6\}, \{a_5, a_7, a_{18}\}, \{a_4, a_8, a_{17}\}, \{a_9, a_{16}\}, \{a_{10}, a_{15}\}, \{a_1, a_{11}\}, \{a_2, a_{12}, a_{14}\}, \{a_3, a_{13}\}$$

See Figure 7.5 for the pictorial representation of $\mathfrak{A}$. In this structure the blocks are,

$$B_1 = \{a_1, a_2, a_3\}$$

$$B_2 = \{a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}\}$$

$$B_3 = \{a_{15}, a_{16}, a_{17}, a_{18}\}$$

The elements a_4, a_{15} are labelled L^+ since their predecessors in the linear order a_3, a_{14} (respectively) are such that $a_4 \ll_p a_3$ and $a_{15} \ll_p a_{14}$. Likewise, elements a_3, a_{14} are labelled R^- since their successors a_4 and a_{15} (resp.) are such that $a_4 \ll_p a_3$ and $a_{15} \ll_p a_{14}$.

A *partial run* of an automaton $\mathcal{A}$ is a pair (p, q) of states of $\mathcal{A}$. Two partial runs $r = (p, q)$ and $s = (q, r)$ can be *concatenated* (or *connected*) to a partial run $t = r \cdot s = (p, r)$. For a partial run $r = (p, q)$, p and q are called *start* and *end* of the run, respectively.

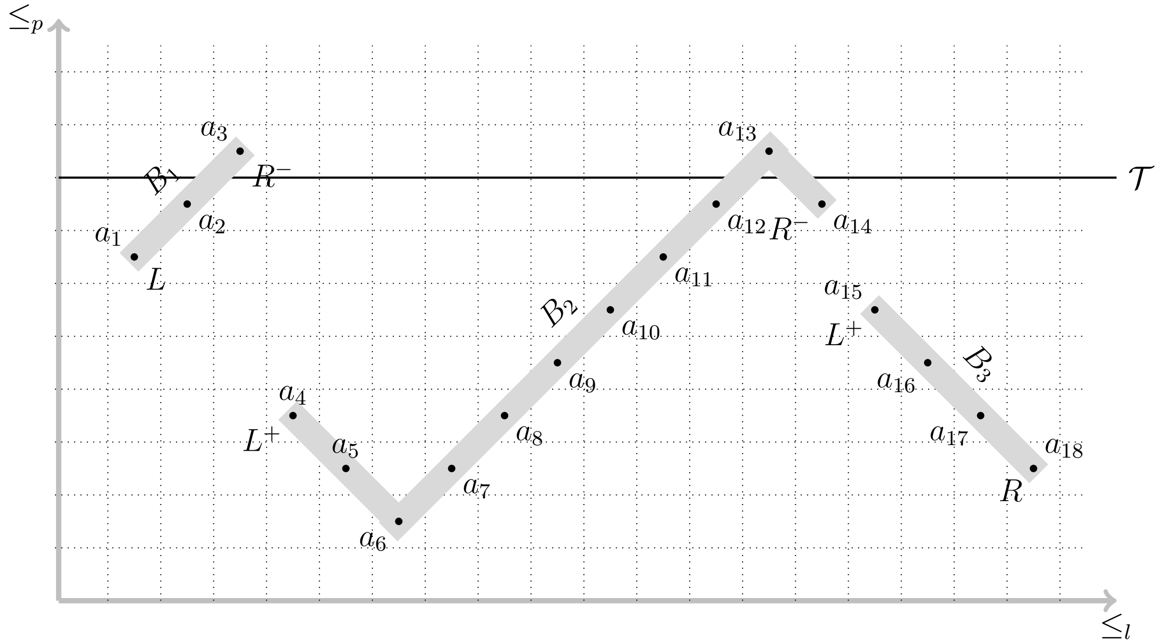

Figure 7.5: Blocks in a snippet of a 3-bounded $(+1_l, +1_p, \leq_p)$-structure.

Lemma 7.3.12. *For a given finite state transducer $\mathcal{B}$ and two states s, t of $\mathcal{B}$, there is a finite state automaton accepting exactly those sequences p over $parikh_k(\Pi)$ for which there is a block $B = u_1, \ldots, u_n$ with*

- *The preorder projection of B is p.*
- *There is a run of $\mathcal{B}$ on B starting in state s and ending in state t.*
- *u_1 and u_n are the only elements labelled by $L \in \{L^+, L^-\}$ and $R \in \{R^+, R^-\}$.*

Proof. We describe how a finite state automaton $\mathcal{A}$ for a finite state transducer $\mathcal{B}$ works on an input sequence $p = p_1, \ldots, p_m$ over $parikh_k(\Pi)$.

The automaton $\mathcal{A}$ successively constructs a run of $\mathcal{B}$ while reading p. Therefore $\mathcal{A}$ stores a multiset of at most k partial runs in its memory. At the beginning no partial runs are stored. Now $\mathcal{A}$ performs, for every input symbol p_i, one round of the following two steps. First, $\mathcal{A}$ guesses for every element u represented by p_i, a partial run (p, q) and a symbol $\sigma \in \Sigma$ such that when $\mathcal{B}$ is in state p and reads σ, it outputs the label of u and goes into state q. If u is marked with $L \in \{L^+, L^-\}$ (or

$R \in \{R^+, R^-\}$), a partial run (s, q) (or (q, t)) is guessed, where q is an arbitrary state from $\mathcal{B}$. In this case the start (end) of this partial run is marked as *dead* and will never be connected in what follows. Secondly, to starts and ends (that are not marked dead) of partial runs stored in the memory, $\mathcal{A}$ connects a partial run obtained in the first step. Note that one partial run obtained in the first step can be connected to none, one or two partial runs from the memory. After every round $\mathcal{A}$ makes sure that at most k partial runs are stored. $\mathcal{A}$ accepts, when only the partial run (s, t) is stored after the last round.

□

7.3.2 When $\leq_p$ is a linear order

We introduce some of the ideas for the k-ODA case by warming up with the 1-ODA case. In 1-bounded $(+1_l, +1_p, \leq_p)$-structures the preorder is a linear order, hence elements from $parikh_1(\Pi)$ can be identified with Π. Further, in the 1-bounded case $L(B)$ and $R(B)$ are the highest and lowest elements with respect to $\leq_p$.

Proposition 7.3.13. *The emptiness problem of* 1*-ODA can be reduced to the emptiness problem of multicounter automata.*

Proof. Given a 1-ODA $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ we construct a multicounter automaton $\mathcal{M}$ such that $\mathcal{A}$ accepts a 1-bounded $(+1_l, +1_p, \leq_p)$-structure if and only if $\mathcal{M}$ accepts a sequence over $parikh_1(\Sigma)$.

For a $(+1_l, +1_p, \leq_p)$-structure $\mathfrak{A}$ with blocks $B_1, \ldots, B_l$ (ordered by the preorder), the automaton $\mathcal{M}$ on input $pp(\mathfrak{A})$ works as follows. While running over $pp(\mathfrak{A})$ it guesses and verifies a run of $\mathcal{C}$. In parallel $\mathcal{M}$ constructs a run of $\mathcal{B}$. In a nutshell this comprises the following two simultaneous steps:

- $\mathcal{M}$ guesses for every block B_i a partial run (s, t) such that $\mathcal{B}$ can reach B_i in state s and leave B_i in state t.
- $\mathcal{M}$ combines those partial runs into one complete run of $\mathcal{B}$.

We make this more precise. The multicounter automaton guesses those positions in $pp(\mathfrak{A})$ in which the blocks $B_1, \ldots, B_l$ start and end.

Note that the linear order projection (string projection) of $\mathfrak{A}$ is a permutation $C_1 = B_{\pi(i_1)}, \ldots, C_l = B_{\pi(i_l)}$ of the blocks $B_1, \ldots, B_l$ where $\pi : [l] \to [l]$ is a permutation of $[l]$, such that the following conditions are met;

Let $j \in [l]$, $C_h = B_{\pi(j)}$, $C_{h-1} = B_{\pi(i)}$ and $C_{h+1} = B_{\pi(k)}$ such that $h - 1 \geq 1$ and $h + 1 \leq l$,

- if the leftmost position of B_j is labelled by L^- then $R(B_i) \ll_p L(B_j)$,
- if the leftmost position of B_j is labelled by L^+ then $L(B_j) \ll_p R(B_i)$.
- if the rightmost position of B_j is labelled by R^- then $L(B_k) \ll_p R(B_j)$,
- if the rightmost position of B_j is labelled by R^+ then $R(B_j) \ll_p L(B_k)$.

These conditions state that the linear order projection (the permutation π) is consistent with the preorder marking. The way the automaton guesses such a permutation is a la the proof of Lemma 6.5.3.

The multicounter works as follows. Everytime the counter automaton guesses that a new block B_j starts, it guesses a partial run (s, t) of $\mathcal{B}$ on B_j and verifies the guess using Lemma 7.3.12. To obtain a complete run of $\mathcal{B}$, the partial runs for the blocks need to be arranged properly according to the L, R markings on the blocks. Therefore $\mathcal{M}$ has counters from the set $Q \times Q$ where, intuitively, the counter (s, t) stores the number of partial runs of $\mathcal{B}$ from s to t that have already been seen during the simulation.

We describe how a complete run of $\mathcal{B}$ is successively constructed, by outlining what happens if $\mathcal{M}$ processes block B_j. Intuitively, the partial run (s, t) is connected to those partial runs that have already been seen. We assume that B_j is neither the first nor the last block, i.e. $j \neq 1$ and $j \neq l$. Further, we assume that the leftmost position u of B_j occurs before the rightmost position v of B_j in the preorder projection. When encountering u, the multicounter automaton $\mathcal{M}$ stores a partial run r in its state, depending on whether u is labelled with L^+ or L^-:

- If u is marked by L^+, the block C_{h-1} did not occur in the preorder projection yet. The partial run $r = (s, t)$ is stored in the state.

- If u is marked by L^-, the block C_{h-1} did already occur in the preorder projection. Therefore $\mathcal{M}$ can guess a partial run (s', s) that ends at the leftmost position of $C_h = B_{\pi(j)}$ and is already saved in the counters. If the last position in the preorder was an R^+ position with partial run (s', s), then $\mathcal{M}$ makes sure that the counter corresponding to (s', s) is at least 2. Now, $\mathcal{M}$ stores the partial run $r = (s', t)$ in its state and decrements the counter (s', s).

When encountering v, block B_j was read completely and a partial run r', that depends on whether v is labelled with R^- or R^+, is saved in the counter r'. The partial run r' is obtained as follows:

- If v is marked by R^+, the block C_{h+1} did not occur in the preorder projection yet. The register for the partial run $r' = r$ is incremented.

- If v is marked by R^-, the block C_{h+1} did already occur in the preorder projection. Therefore $\mathcal{M}$ can guess a partial run (t, t') that starts at the rightmost position of $C_h = B_{\pi(j)}$ and is already saved in the counters. If the last position in the preorder was an L^- position with partial run (t, t'), then $\mathcal{M}$ makes sure that the counter corresponding to (t, t') is at least 2. The register for the partial run $r' = r(t, t')$ is incremented and the register for (t, t') is decremented. (Here, $r(t, t')$ denotes the concatenation of the two partial runs.)

The cases where B_j is the first or last blocks as well as the case when $\mathcal{M}$ encounters the rightmost position v of B_j first can be settled similarly.

We claim that $L(\mathcal{A})$ is non-empty if and only if $L(M)$ is nonempty.

For the left to right direction, assume that M has a successful run on a sequence of Blocks $B_1, \ldots, B_n$. This implies that there is a successful run of C on $B_1, \ldots B_n$. What remains to be shown is that $\mathcal{B}$ has a run on a permutation of $B_1, \ldots, B_n$ which is

$$C_1 = B_{\pi(i_1)}, \ldots, C_n = B_{\pi(i_n)}$$

where $\pi : [n] \to [n]$ is the permutation and π is consistent with the marking on the blocks.

For $1 \leq j \leq n$, $\pi_j : [j] \to [n]$ be an injective function. We call π_j a partial permutation. Given $\mathsf{Range}(\pi_j)$ it can be partitioned into $\mathsf{Range}(\pi_j) = \bigcup_{1\leq i\leq k} s_i$ for some $k \leq j$, where each $s_i \subseteq \mathsf{Range}(\pi_j)$ is a maximal interval in $\mathsf{Range}(\pi_j)$. For a maximal interval $s_i = [l, u]$, let $\pi^{-1}(s_i)$ be the sequence

$$\pi^{-1}(s_i) = \pi^{-1}(l), \pi^{-1}(l+1), \dots, \pi^{-1}(u).$$

We call the set of sequences $\{\pi^{-1}(s_i) \mid 1 \leq i \leq k\}$ the sequence representation of π_j, denoted by $\mathsf{seq}(\pi_j)$. Define the following equivalence relation $\sim_j$ on the set of partial permutations from $[j]$ to $[n]$ as follows. Given $\pi : [j] \to [n]$ and $\pi' : [j] \to [n]$ $\pi \sim_j \pi'$ if $\mathsf{Seq}(\pi) = \mathsf{Seq}(\pi')$. Given a sequence $s =< i_1, \dots, i_k >$ and $i \in \mathbb{N}$, we denote by $s \cdot i$ the sequence $< i_1, \dots, i_k, i >$.

Given $B_1, \dots, B_j$ a partial permutation π_j, the concaternation of $B_{i_0} \dots B_{i_k}$ where $s =< i_0, \dots, i_k >\in \mathsf{Seq}(\pi_j)$ is called the maximal segment defined by s in π_j.

We claim that when the automaton has finished reading the block B_j then there is a partial permutation (partial function) $\pi_j : [j] \to [n]$ of blocks $B_1 \dots B_j$ consistent with the markings such that (1) if the counter (s, t) is k, there are k maximal segments defined by π_j taking $\mathcal{B}$ from s to t. (2) if the partial permutation $\pi'_j : [j] \to [n]$ is such that $\pi_j \sim_j \pi'_j$ then in the permutation π'_j of the blocks $B_1, \dots, B_j$ there are k maximal segments defined by π'_{j+1} taking $\mathcal{B}$ from s to t.

Observe that if the permutation π_j of $B_1, \dots, B_j$ is consistent with the marking then any other permutation π'_j of $B_1, \dots, B_j$ is also consistent with the marking.

The claim implies that $\mathcal{B}$ has an accepting run since the claim implies a permutation $\pi : [n] \to [n]$ of $B_1, \dots, B_n$ which is consistent with the marking and on which $\mathcal{B}$ has a successful run.

We prove the claim using induction on j. The base step is trivial. For the inductive step assume that the claim is true on the j-th step. Assume that the partial run corresponding to B_{j+1} is verified to be (s, t) and the leftmost and rightmost positions of B_{j+1} are labelled by L^- and R^+ respectively. In this case the automaton nondeterministically chooses a pair (s', s).

Assume that the leftmost position of B_{j+1} was not preceded by the rightmost position of B_i. The automaton decreases the counter (s', s) and increases a counter

(s', t). Observe that since counter (s', s) is non-empty, by induction hypothesis, there is a maximal segment (corresponding to say the sequence $s_i \in \mathsf{Seq}\,(\pi_j)$) defined by π_j let which corresponds to a run from s' to s. We define $\mathsf{Seq}\,(\pi_{j+1}) = \mathsf{Seq}\,(\pi_j) - \{s_i\} \cup \{s_i \cdot (j+1)\}$. The fact that there is such a π_{j+1} is guaranteed by the definition of Seq and the fact that $j+1 \leq n$. Observe that claim (1) follows from the fact that all segments except the one corresponding to s_i are untouched (and their respective $(Q \times Q)$-counts), and the the count of segments having run (s', t) increased by one (by the addition of $s_i \cdot (j+1)\}$) and the count of segments having run (s', s) decreased by one (by the deletion of s_i). This change is reflected in the counters. Claim (2) follows trivially by definition.

Consider the case when the leftmost position of B_{j+1} was preceded by the rightmost position of B_j which was labelled by R^+ and B_j had a partial run (s', s). In this case the automaton decreases the counter twice. This ensures that there is a sequence in $\mathsf{Seq}(\pi_j)$ which does not end in j but corresponds to a maximal segment which has a run from s' to s. We proceed by adding $(j+1)$ to this sequence and repeat the above argument. Thus we preserve the consistency of π_{j+1} with respect to the marking.

The case when B_{j+1} has its leftmost and rightmost positions are marked by R^- and L^+ is symmetric.

When the leftmost and rightmost positions of B_{j+1} is L^- and R^-, then The automaton decreases two counters (s', s), (t, t') nondeterministically and increases the counter (s', t'). By induction hypothesis, there are two sequences s_i, s_k in $\mathsf{Seq}(\pi_j)$ whose segments have runs from s' to s and t to t'. We define the sequence $\mathsf{Seq}\,(\pi_{j+1}) = \mathsf{Seq}\,(\pi_j) - \{s_i, s_k\} \cup \{s_i \cdot (j+1) \cdot s_k\}$. The argument is similar to the previous case.

When the leftmost and rightmost positions of B_{j+1} is L^+ and R^+, then The automaton increases the counter (s, t). This corresponds to defining $\mathsf{Seq}\,(\pi_{j+1}) = \mathsf{Seq}\,(\pi_j) \cup \{< (j+1) >\}$. In this case the number of segments corresponding to the run (s, t) increases by one which is reflected in the counter.

To show that if $\mathfrak{A}$ is accepted by $\mathcal{A}$ then there is a sequence of blocks $B_1, \ldots, B_n$ accepted by the multicounter automata, we take $w = B_1, \ldots, B_n$ as the blocks in the preorder projection of $\mathfrak{A}'$ where $\mathfrak{A}'$ is the relabelling of $\mathfrak{A}$ by the transducer $\mathcal{B}$.

Observe that $\mathcal{C}$ has a successful run over w. We know that there is a permutation $\pi : [n] \to [n]$ of w such that $\mathcal{B}$ has a successful run on $\pi(w)$. To show that the mutlicounter automaton has a successful run we proceed as follows. Define the counter configurations of the automaton after the j-th step as; the counter (s, t) carry the value k if there are k-maximal segments defined by $\mathsf{Seq}(\pi_j)$ where π_j is π restricted to $[j]$. The automaton chooses a transition based on the marking as in the proof of Lemma 6.5.3.

□

From this we can conclude that,

Theorem 7.3.14. *Finite satisfiability of* $\mathrm{FO}^2(\Sigma, \leq_{l_1}, +1_{l_1}, +1_{l_2})$ *is decidable.*

7.3.3 When $k > 1$.

For k-bounded preorders with $k > 1$ we extend the construction of Proposition 7.3.13. However, this situation is more complicated for several reasons:

- While reading the preorder projection, the multicounter automaton has to process several blocks at once.
- The L and R markers can appear anywhere in the preorder projection of a block.
- The interaction L and R markers of several blocks has to be considered.

Those problems can be solved.

Theorem 7.3.15. *The emptiness problem of k-ODA can be reduced to the emptiness problem of multicounter automata.*

Proof. Again, for a given k-ODA $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ we construct a multicounter automaton $\mathcal{M}$ such that $\mathcal{A}$ accepts a k-bounded $(+1_l, +1_p, \leq_p)$-structure if and only if $\mathcal{M}$ accepts a sequence over $parikh_k(\Sigma)$.

Consider an input $(+1_l, +1_p, \leq_p)$-structure $\mathfrak{A}$ with blocks $B_1, \ldots, B_l$ (ordered by the linear order). Upto k blocks can now overlap in the preorder projection (see

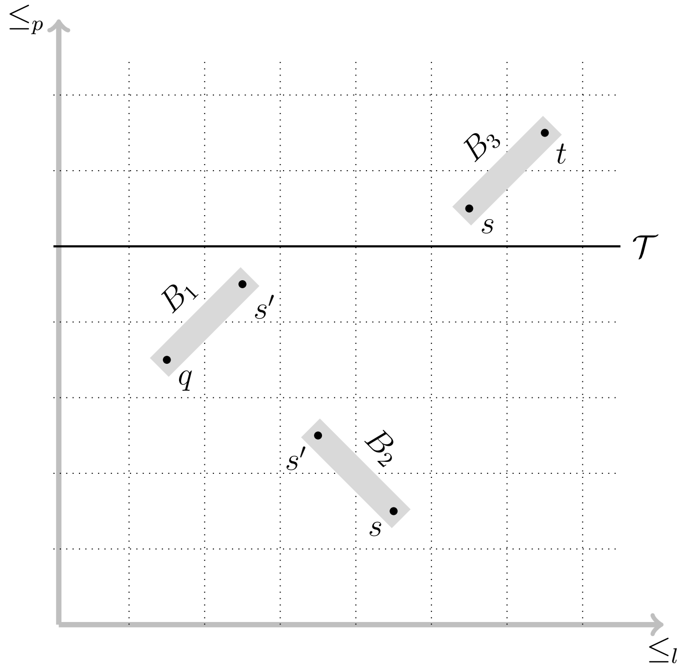

Figure 7.6: How a 1-ODA is simulated by a multicounter automaton $\mathcal{M}$. When $\mathcal{M}$ reaches the solid line $\mathcal{T}$, the counter for (q, s) is one. When starting to read block B_3, the counter for (q, s) is decremented and (q, t) is stored in the state.

e.g. line $\mathcal{T}$ in Figure 7.5). Blocks whose start has been read, but whose end still needs to be read, are called *active*. Thus there are at most k active blocks. The automaton $\mathcal{M}$ guesses and verifies a run of $\mathcal{C}$ on $pp(\mathfrak{A})$. In parallel $\mathcal{M}$ constructs a run of $\mathcal{B}$. Here, this comprises the following simultaneous steps:

i) A symbol $p \in parikh_k(\Sigma)$ is partitioned corresponding to the active blocks.

ii) $\mathcal{M}$ guesses for every newly started active block B_i a partial run (s, t) such that $\mathcal{B}$ can reach B_i in state s and leave B_i in state t.

iii) $\mathcal{M}$ combines those partial runs into one complete run of $\mathcal{B}$.

We make this more precise. For verifying ii), a subautomaton $\mathcal{M}_i$ is assigned to every active block A_i. $\mathcal{M}_i$ can check ii) using Lemma 7.3.12. More precisely, when reading $p \in parikh_k(\Sigma)$, the automaton $\mathcal{M}$ guesses a partition P of p into at most k subvectors that sum up to p. For each $q \in P$ it guesses whether q belongs to some active block or starts a new block. If q belongs to an active block A_i, then $\mathcal{M}_i$ gets p as input for this step. In case q starts a new active block B_i, a partial run (s,t) of $\mathcal{B}$ is guessed such that $\mathcal{B}$ reaches B_i in s and leaves B_i in t. A new subautomaton is created to verify the partial run, using Lemma 7.3.12. Active blocks that do not have a $q \in P$ are closed. (As there are at most k active blocks, there are at most k subautomata running simultaneously.)

To obtain a complete run of $\mathcal{B}$, the partial runs for the blocks need to be arranged properly. As before $\mathcal{M}$ has counters from the set $Q \times Q$ saving the number of partial runs of $\mathcal{B}$ from s to t that have already been seen during the simulation. However, some partial runs will be *cached* in the states of $\mathcal{M}$, namely runs where one, either start or end state corresponds to an active block. For every cached run r, two pointers $L(r)$ and $R(r)$ are stored in the state of $\mathcal{M}$ that contain the active block that corresponds to the start or end state of r, respective, or $'-'$ if there is no corresponding active block. (Since there are at most k active blocks, such pointers can be stored.)

Furthermore, $\mathcal{M}$ saves, for every cached run, which of its end points can be connected at the moment and remembers for every partial run r how many have been added in the last round due to encountering R and how many due to L.

We describe how a complete run of $\mathcal{B}$ is successively constructed, by outlining what happens if $\mathcal{M}$ processes block B_i. Intuitively, the partial run (s,t) will be connected to those partial runs that have already been seen. We assume that B_i is neither the first nor the last block, i.e. $i \neq 1$ and $i \neq l$. Further, we assume that the leftmost position u of B_i occurs before the rightmost position v of B_i in the preorder projection.

When encountering u, the multicounter automaton $\mathcal{M}$ proceeds as follows

- If u is marked by L^+, the block B_{i-1} did not occur in the preorder projection yet. The partial run $r = (s,t)$ is cached in the state with $R(r) = B_i$. Furthermore $L(r)$ is marked as non-connectable for the next step.

- If u is marked by L^-, the block B_{i-1} did already occur in the preorder projection. Now, $\mathcal{M}$ guesses whether the run q corresponding to B_{i-1} is cached or saved in the counter q:

- If q is cached and $R(q)$ is connectable, then q is replaced by the partial run $r' = q \cdot r$ with $L(r') = L(q)$ and $R(r') = B_i$.
- If q is saved in q and counter q is connectable, then q is replaced by the partial run $r' = q \cdot r$ with $L(r') =' -'$ and $R(r') = B_i$.

When encountering v, the multicounter automaton $\mathcal{M}$ proceeds as follows:

- If v is marked by R^+, the block B_{i+1} did not occur in the preorder projection yet. The register for the partial run $r' = r$ is incremented.
- If v is marked by R^-, the block B_{i+1} did already occur in the preorder projection. Therefore $\mathcal{M}$ can guess a partial run (t, t') that starts at the rightmost position of B_i and is already saved in the counters. If the last position in the preorder was an L^- position with partial run (s', s), then $\mathcal{M}$ makes sure that the counter corresponding to (t, t') is at least 2. The register for the partial run $r' = r(t, t')$ is incremented and the register for (t, t') is decremented. (Here, $r(t, t')$ denotes the concatenation of the two partial runs.)

The cases where B_i is the first or last blocks as well as the case when $\mathcal{M}$ encounters the rightmost position v of B_i first can be settled similarly. □

From this we conclude that,

Theorem 7.3.16. *Finite satisfiability of* $\mathrm{FO}^2(\Sigma, +1_{l_1}, \leq_{p_1}, +1_{p_1})$ *on k-bounded ordered data words is decidable.*

7.3.4 A Hardness Result for $\mathrm{FO}^2(\leq_{l_1}, +1_{l_1}, +1_{l_2})$

In this section we prove a matching lower bound for $\mathrm{FO}^2(+1_l, +1_p, \leq_p)$ over k-bounded structures.

Proposition 7.3.17. *Finite satisfiability of* $\mathrm{FO}^2(\leq_{l_1}, +1_{l_1}, +1_{l_2})$ *is at least as hard as the reachability problem for vector addition systems.*

Proof. We reduce the non-emptiness of multicounter automata to satisfiability of $\mathrm{FO}^2(\leq_{l_1}, +1_{l_1}, +1_{l_2})$.

Let M be a multicounter automaton with counters $C = \{1, \ldots m\}$ and state set Q. Transitions $\Delta = \{\delta_1, \ldots, \delta_l\}$ are from the language $Q \times \{D_j \mid j \in C\}^* \times \{I_j \mid j \in C\}^* \times Q$, where D_j and I_j stand for decrementing and incrementing the counter j, respective.

We write a sentence φ in $\mathrm{FO}^2(\leq_{l_1}, +1_{l_1}; +1_{l_2})$ which ensures the following:

- The string projection of the order $\leq_{l_1}$ is of the form $\delta_{i_1} \ldots \delta_{i_n}$ such that δ_{i_1} is a transition from the initial state, δ_{i_n} is a transition to a final state and every two successive transitions have a common state.
- The string projection of the order $\leq_{l_2}$ is in the language $Q^*(I_1 D_1 + \ldots + I_k D_k)^*$.
- It is the case that $\bigwedge_{j \in C} \forall x \forall y \, [(I_j(x) \wedge D_j(y) \wedge +1_{l_2}(x, y)) \rightarrow x \leq_{l_1} y]$.

□

Since the logic $\mathrm{FO}^2(+1_l, +1_p, \leq_p)$ allows for axiomatizing 1-boundedness, we have the following corollary.

Corollary 7.3.18. *Finite satisfiability of* $\mathrm{FO}^2(+1_l, +1_p, \leq_p)$ *over k-bounded ordered data words is at least as hard as the reachability problem for vector addition systems.*

7.4 Discussion

In this chapter we showed that finite satisfiability problems of $\mathrm{FO}^2\,(\Sigma, \leq_{l_1}, +1_{l_1}, +1_{l_2})$ and $\mathrm{FO}^2\,(\Sigma, +1_{l_1}, +1_{p_2}, \leq_{p_2})$ on k-bounded structures are decidable. The automata theoretic proof is a sophisticated version of the techniques used in Chapter 6. However the most important question is whether the restriction of k-boundedness can be removed preserving decidability.

8 Conclusion

8.1 Remarks on automata for data words

We saw that finite state automata augmented with counters, namely CCA, can give us a reasonably good automaton model for data words. They fall (roughly) in between register automata and class-memory automata in terms of expressive power and complexity of decision problems. Further, CCA can be strengthened to match the expressive power of class memory automata. The main attraction of this automaton is its comparatively lower complexity of its decision problems. However we do not see any differences between these three automata in terms of complexity of model checking. Moreover none of the automata are closed under complementation. Further their deterministic versions are strictly weaker compared to the nondeterministic counterparts.

In the beginning we mentioned that one of the important questions regarding data words is on the notion of regularity for data words, the notion which guarantees low complexity decision problems, good expressive power and nice closure properties. However, none of these automata can be called regular in the true sense of the word, an opinion which is also shared by the community [BS10].

However, the studies so far have revealed the difficulty of the problem we are dealing with. We saw that there is a natural connection between register automata and finite state automata in terms of reachability problem. Similarly, there is a natural correspondence between CCA (as well as CMA) and vector addition systems.

Thus these automata mirror the known machine models for general infinite-state systems for the case of data words. Hence the quest for regularity for data words resonates well with the quest for expressive yet easily analyzable infinite-state systems – one that continues on.

8.2 Remarks on logics

We saw that $\mathrm{FO}^2(\Sigma, +1_{l_1}, +1_{l_2})$ is elementarily decidable. However, the current decision procedure for $\mathrm{FO}^2(\Sigma, +1_l, \leq_p, +1_p)$ is not of elementary complexity. This is further worsened by the fact that reachability in vector addition systems – a notoriously difficult problem with no elementary decision procedure known – reduces to this logic. The high complexity of the satisfiability problem reduces heavily the applicability of these logics in the present scenario. Secondly, the fragments themselves are quite restrictive, the restriction of two variables and absence of order relations severely affects the kinds of properties expressible in this logic.

However the theoretical importance of these fragments should not be underestimated, especially as part of classifying the decidable fragments of first order logic (called the classification problem). We conclude by noting the current status of research on two-variable logics with additional successor and order relations (Figure 8.1).

Logic	**Complexity** (lower/upper)	**Comments**
	One linear order	
$FO^2(+1_l)$	NEXPTIME-complete	[EVW02]
$FO^2(\leq_l)$	NEXPTIME-complete	[EVW02]
$FO^2(+1_l, \leq_l)$	NEXPTIME-complete	[EVW02]
	One total preorder	
$FO^2(+1_p)$	NEXPTIME-complete	
$FO^2(\leq_p)$	NEXPTIME-complete	
$FO^2(+1_p, \leq_p)$	EXPSPACE-complete	[SZ11]
	Two linear orders	
$FO^2(+1_{l_1}; +1_{l_2})$	NEXPTIME/2-NEXPTIME	[Man10]
$FO^2(+1_{l_1}; \leq_{l_2})$	NEXPTIME/EXPSPACE	[SZ11]
$FO^2(+1_{l_1}, \leq_{l_1}; +1_{l_2})$	VASS-REACHABILITY/Decidable	Proposition 7.3.17
$FO^2(+1_{l_1}, \leq_{l_1}; \leq_{l_2})$	NXPTIME/EXPSPACE	[SZ11]
$FO^2(+1_{l_1}, \leq_{l_1}$ $; +1_{l_2}, \leq_{l_2})$	Undecidable	Proposition 5.4.6
	Two total preorders	
$FO^2(+1_{p_1}, +1_{p_2})$	Undecidable	[MZ11]
$FO^2(+1_{p_1}; \leq_{p_2})$	Undecidable	[MZ11]
$FO^2(\leq_{p_1}; \leq_{p_2})$	Undecidable	[SZ10]
	One linear order and one total preorder	
$FO^2(+1_{l_1}; +1_{p_2})$	?	
$FO^2(+1_{l_1}, \leq_{l_1}; +1_{p_2})$	Undecidable	[MZ11]
$FO^2(+1_{l_1}, \leq_{l_1}; \leq_{p_2})$	Undecidable	[BDM+11]
$FO^2(+1_{l_1}; +1_{p_2}, \leq_{p_2})$	?	
$FO^2(\leq_{l_1}; +1_{p_2}, \leq_{p_2})$	EXPSPACE-complete	[SZ11]
	Many orders	
$FO^2(\leq_{l_1}, \leq_{l_2}, \leq_{p_3})$	Undecidable	[SZ10]
$FO^2(\leq_{l_1}, \ldots, \leq_{l_3})$	Undecidable	[Kie11]
$FO^2(+1_{l_1}, \ldots, +1_{l_k})$	?	

Figure 8.1: Summary of results on finite satisfiability of FO^2 with successor and order relations. Cases that are symmetric and where undecidability is implied are omitted.

Bibliography

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BDM+11] Mikolaj Bojanczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27, 2011.

[BS10] Henrik Björklund and Thomas Schwentick. On notions of regularity for data languages. *Theoretical Computer Science*, 411(4-5):702–715, 2010.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[DL09] Stéphane Demri and Ranko Lazic. LTL with the freeze quantifier and register automata. *ACM Transactions in Computational Logic*, 10(3):16:1–16:30, 2009.

[End72] Herbert B. Enderton. *A mathematical introduction to logic*. Academic Press, 1972.

[Esp96] Javier Esparza. Decidability and complexity of petri net problems - an introduction. In *Advanced Courses*, pages 374–428, 1996.

[EVW02] Kousha Etessami, Moshe Y. Vardi, and Thomas Wilke. First-order logic with two variables and unary temporal logic. *Information and Computation*, 179(2):279 – 295, 2002.

[GKV97] Erich Grädel, Phokion G. Kolaitis, and Moshe Y. Vardi. On the decision problem for two-variable first-order logic. *Bulletin of Symbolic Logic*, 3(1):53–69, 1997.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[KF94] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[Kie11] Emanuel Kieronski. Decidability issues for two-variable logics with several linear orders. In *CSL*, volume 12 of *LIPIcs*, pages 337–351, 2011.

[KO05] Emanuel Kieronski and Martin Otto. Small substructures and decidability issues for first-order logic with two variables. In *LICS*, pages 448–457, 2005.

[Kos82] S. Rao Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *STOC*, pages 267–281. ACM, 1982.

[KT09] Emanuel Kieronski and Lidia Tendera. On finite satisfiability of two-variable first-order logic with equivalence relations. In *LICS*, pages 123–132, 2009.

[Lip76] R. J. Lipton. The reachability problem requires exponential space. Technical report 62, Department of Computer Science, Yale University, 1976.

[Man10] Amaldev Manuel. Two orders and two variables. In *MFCS*, volume 6281 of *LNCS*, pages 513–524, 2010.

[May81] Ernst W. Mayr. An algorithm for the general petri net reachability problem. In *STOC '81*, pages 238–246. ACM, 1981.

[Mor75] M. Mortimer. On languages with two variables. *Zeitschr. f. math. Logik u. Grundlagen d. Math.*, 21:135–140, 1975.

[MR11] Amaldev Manuel and R. Ramanujam. Class counting automata on datawords. *Int. J. Found. Comput. Sci.*, 22(4):863–882, 2011.

[MRR+08] Anca Muscholl, R. Ramanujam, Michaël Rusinowitch, Thomas Schwentick, and Victor Vianu. *Beyond the Finite: New Challenges in Verification and Semistructured Data.* Dagstuhl Seminar 08171 Proceedings. 2008.

[MZ11] Amaldev Manuel and Thomas Zeume. Two variable logic with a linear successor and a preorder. Under preparation, 2011.

[NSV04] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions in Computational Logic*, 5(3):403–435, 2004.

[Ott01] Martin Otto. Two variable first-order logic over ordered domains. *Journal of Symbolic Logic*, 66(2):685–702, 2001.

[Sco62] D. Scott. A decision method for validity of sentences in two variables. *Journal of Symbolic Logic*, 27:377, 1962.

[SI00] Hiroshi Sakamoto and Daisuke Ikeda. Intractability of decision problems for finite-memory automata. *Theoretical Computer Science*, 231(2):297–308, 2000.

[ST11] Luc Segoufin and Szymon Torunczyk. Automata based verification over linearly ordered data domains. In *STACS*, volume 9 of *LIPIcs*, pages 81–92, 2011.

[SZ10] Thomas Schwentick and Thomas Zeume. Two-variable logic with two order relations. In *CSL*, volume 6247 of *LNCS*, pages 499–513, 2010.

[SZ11] Thomas Schwentick and Thomas Zeume. Two-variable logic with two order relations. To appear, 2011.

[Zei06] Daniel Zeitlin. Look-ahead finite-memory automata. Master's thesis, Technion - Israel Institute of Technology, July 2006.