BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FORMAL MODELS FOR DATA LANGUAGES

FORMÁLNÍ MODELY PRO PRÁCI S DATOVÝMI JAZYKY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **JAN VAŠÁK**
AUTOR PRÁCE

SUPERVISOR **Ing. ONDŘEJ LENGÁL, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

154542

Institut:	Department of Intelligent Systems (DITS)
Student:	**Vašák Jan**
Programme:	Information Technology
Title:	**Formal Models for Data Languages**
Category:	Theoretical Computer Science
Academic year:	2023/24

Assignment:

1. Study the theory of formal models for data languages, such as variants of register automata or logics. Focus on models that are suitable for an implementation of matching regular expressions with back-references or modeling programs with lists and sets.
2. Based on a discussion with the supervisor, choose at least one of the following options:
    1. Study the theoretical properties of the considered models, e.g. decidability and complexity of problems such as emptiness, language inclusion, or functional equivalence.
    2. Develop efficient algorithms for working with a chosen model, focusing on efficient handling of operations such as inclusion testing, reduction, etc.
3. Execute the option chosen in item (2).
4. In the case the second option of (2) was chosen, implement the developed algorithms and compare them experimentally with algorithms that work on finite automata (for instance over an alphabet such as ASCII or Unicode).
5. Discuss the obtained results and possible further extensions.

Literature:

- Michael Kaminski, Nissim Francez: Finite-Memory Automata. Theor. Comput. Sci. 134(2): 329-363 (1994)
- Stéphane Demri, Ranko Lazic: LTL with the freeze quantifier and register automata. ACM Trans. Comput. Log. 10(3): 16:1-16:30 (2009)
- Diego Figueira: Alternating register automata on finite words and trees. Log. Methods Comput. Sci. 8(1) (2012)
- Gulčíková, S. and Lengál, O., 2022. Register Set Automata (Technical Report). *arXiv preprint arXiv:2205.12114*.
- N. Tzevelekos and R. Grigore, “History-Register Automata,” in *Foundations of Software Science and Computation Structures*, F. Pfenning, Ed., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 17–33. doi: 10.1007/978-3-642-37075-5_2.

Requirements for the semestral defence:
Item 1.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Lengál Ondřej, Ing., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	6.11.2023

Abstract

Data words are a common way to formally work with words over infinite alphabets. In practice, an infinite alphabet can represent an actually infinite set, such as the integers or a set of strings, or a large finite set, such as the Unicode symbols. We explore some theoretical properties of register set automata, a data word model that, crucially, can be used as a means to determinise a large class of register automata (this allows, e.g., for a deterministic automata representation of a class of regexes with back-references). We also extend streaming data string transducers, a model designed to represent a class of list-processing programs, with set-registers. This extension can, for example, represent a program that removes duplicates from a list, which is not representable using the base model. We then show that this extension's functional equivalence problem is decidable. Lastly, a prototype regex matcher based on register set automata was implemented, and we experimentally show that it performs well under regular expression denial of service attacks that can cripple other matchers used in practice.

Abstrakt

Datová slova jsou běžně používaná pro formální práci se slovy nad nekonečnými abecedami. V praxi může nekonečná abeceda modelovat skutečně nekonečnou množinu, např. celá čísla nebo množinu řetězců, nebo velkou konečnou množinu, jako např. znaky sady Unicode. Tato práce se nejprve věnuje teoretickým vlastnostem registrově množinových automatů. Registrově množinové automaty jsou modelem nad datovými slovy, který lze použít pro determinizaci velké třídy registrových automatů (toto např. umožňuje deterministickou automatovou reprezentaci třídy regulárních výrazů se zpětnými odkazy). Dále jsme rozšířili streaming data string převaděče, model určený pro modelování třídy programů pro zpracování lineárních seznamů, o množinové registry. Toto rozšíření umožňuje např. modelovat program, který odstraní duplicitní hodnoty z lineárního seznamu, což není možné modelovat základními streaming data string převaděči. Ukážeme, že problém funkční ekvivalence je pro toto rozšíření rozhodnutelný. Také byl naimplementován prototyp regex matcheru založený na registrově množinových automatech. Ukážeme, že prototyp si vede dobře pod ReDoS (regular expression denial of service) útoky, které jsou efektivní vůči regex matcherům používaným v praxi.

Keywords

data words, register set automata, streaming data string transducers, regular expressions with back-references

Klíčová slova

datová slova, registrově množinové automaty, streaming data string převodníky, regulární výrazy se zpětnými odkazy

Reference

VAŠÁK, Jan. *Formal Models for Data Languages.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondřej Lengál, Ph.D.

Rozšířený abstrakt

Datová slova jsou běžným formalismem používaným při práci se slovy nad nekonečnými abecedami. Nekonečná abeceda může v praxi reprezentovat nekonečnou množinu (např. celá čísla) nebo velkou konečnou množinu (např. symboly sady Unicode). Datové slovo je sekvence dvojic skládajících se ze symbolu konečné abecedy Σ a datové hodnoty ze spočetně nekonečné datové domény $\mathbb{D}$. Například $(a,1)(b,2)(a,42)$ je datové slovo nad $\Sigma = \{a,b\}$ a $\mathbb{D} = \mathbb{N}$.

Registrové automaty (RA) jsou modelem pro práci s datovými slovy. Rozšiřují konečné automaty o konečnou množinu registrů. Každý registr může obsahovat maximálně jednu datovou hodnotu a hodnoty registrů lze porovnávat na přechodech automatu se vstupní datovou hodnotou (značenou *in*). RA lze využít např. při modelování regulárních výrazů se zpětnými odkazy, nebo verifikaci programů. RA ovšem nelze obecně determinizovat, což omezuje jejich praktické využití například právě pro efektivní porovnávání regulárních výrazů.

Registrově množinové automaty (RsA) jsou model podobný RA, ovšem registry v RsA mohou obsahovat neomezeně velkou množinu hodnot. Na přechodech lze testovat příslušnost *in* do těchto množinových registrů. Klíčová vlastnost RsA je, že je lze použít pro determinizaci určité třídy RA.

Streaming data string převaděče (SDST) jsou model využívající registry jako v registrovém automatu. Navíc jsou vybaveny řetězcovými proměnnými, které jsou využity pro generování výstupu. SDST byly navrženy pro modelování programů zpracovávajících lineární seznamy jedním průchodem.

V této práci jsou zkoumány některé teoretické vlastnosti RsA. Konktrétně se jedná o parametrizaci jejich problému prázdnosti na počtu registrů a porovnání jejich vyjadřovací síly s history registrovými automaty (HRA), modelem podobným RsA, který manipuluje se svými množinovými registry jiným způsobem. Bylo zjištěno, že problém prázdnosti RsA je NL-úplný při omezení na jeden registr a je v $\mathbf{F}_{2^n+1}$ pro n registrů. Dále bylo dokázáno, že deterministické RsA mají větší vyjadřovací sílu než deterministické HRA a že HRA lze vždy převést na RsA. Není ovšem jasné, zda je to možné i naopak.

Dále je v této práci představeno rozšíření SDST o typ množinových registrů. Toto rozšíření umožňuje např. modelování programu, co odstraní duplicitní hodnoty v lineárním seznamu (takový program nelze reprezentovat SDST bez rozšíření). Poté bylo ukázáno, že funkční ekvivalence tohoto rozšíření je rozhodnutelná (což je důležitý výsledek pro použití modelu pro verifikaci a analýzu programů).

V existujícím algoritmu pro determinizaci RA do deterministických RsA byly nalezeny omezení, kdy algoritmus nevygeneruje RsA pro RA, které lze pomocí RsA determinizovat. Pro vyřešení nalezených omezení byly představeny dva doplňující algoritmy. První předzpracovávající RA před determinizací, a druhý upravující detekování nadaproximace původního RA po determinizaci.

Nakonec byl naimplementován prototyp regex matcheru založený na RsA. Prototyp funguje tak, že z regulárního výrazu se zpětnými odkazy vytvoří RA (pokud to je možné) a následně RA determinizuje. Pokud se determinizace podaří, pak lze pomocí vytvořeného RsA přímo porovnávat vstupní řetězce s regulárním výrazem. Tento prototyp byl pak experimentálně porovnán s jinými matchery používanými v praxi pod vygenerovanými ReDoS (regular expression denial of service) útoky. Tyto útoky byly cíleny na regulární výrazy se zpětnými odkazy, které byly použité v praxi. Úspěšně byla determinizována více než třetina regulárních výrazů na které bylo útočeno. Prototyp útočné vstupy porovnával s

determinizovanými regulárními výrazy efektivně, narozdíl od některých matcherů používaných v praxi.

V budoucnu je cílem zjistit, zda SDST rozšíření lze dále rozšířit při zachování rozhodnutelnosti funkční ekvivalence. Dalším plánem je pokusit se rozšířit třídu determinizovatelných RA do RsA. Nakonec je cílem začít pracovat na ReDoS generátoru, který by cílil přímo na regulární výrazy se zpětnými odkazy.

Formal Models for Data Languages

Declaration

I hereby declare that this Bachelor’s thesis was prepared as an original work by the author under the supervision of Ing. Ondřej Lengál, Ph.D. I have listed all the literary sources, publications and other sources that were used during the preparation of this thesis.

. . . .
Jan Vašák
May 7, 2024

Acknowledgements

I would like to thank my supervisor Ing. Ondřej Lengál, Ph.D. for his guidance, patience, and help while working on this thesis. I would also like to thank to all my family and friends who have supported me throughout the writing of this thesis.

Contents

Chapter 1

Introduction

Finite automata are a staple formal model for simple computations. They consist of a finite number of *states* and *transitions*. The input of a finite automaton is a word: a sequence of *symbols* (characters) from a finite set called the *alphabet*. The automaton reads one input symbol at a time and decides on a new state based on the transitions available to it in its current state as well as the read symbol (transitions have an origin state, a symbol, and a destination state). Automata have a wide range of applications, common examples include text processing and compiler design. Famously, finite automata are equivalent to *regular expressions* (regexes), a formal system for describing *regular languages* (sets of words accepted by some finite automaton).

Finite automata are only equipped to work with words over finite alphabets, as they only have a finite number of transitions, each with just one symbol on it. However, sometimes one might want to model an infinite alphabet (e.g. when modelling integers), or a finite alphabet so large that it is better to treat it as infinite (e.g., Unicode symbols). Data languages are a common way to formally work with words over infinite alphabets.

Data languages are sets of *data words*, finite sequences made up of pairs consisting of a symbol from a finite alphabet and a data value from a countably infinite *data domain*. Formal models that work over data words commonly utilize a finite number of some form of registers — a place to store a data value in order to refer to it later. In this work, we will focus on models suitable for pattern matching with back-references and modelling list-processing programs.

Register automata, first proposed in [16], extend finite automata with registers, each storing a single data value. They are useful in, e.g., modelling regular expressions with back-references, program verification [9], or malware specification [26]. Register automata, however, are not determinisable in general, which makes them less useful in some implementations (e.g. regex matching).

Register set automata, proposed in [11], are an extension of register automata. Unlike register automata, they allow for a set of values to be stored in each register. This allows for an algorithm to be able to determinise a large class of non-deterministic register automata into deterministic register set automata, using a similar principle to the classical subset construction algorithm for determinising finite automata [21]. Due to this property, register set automata are an interesting model when it comes to automata-based matching of patterns with back-references. We will also examine the relationship of register set automata and history register automata, a similar model proposed earlier in [10].

Streaming data string transducers, proposed in [1], are a transducer model (a transducer is essentially an automaton with an output). They are equipped with a set of data variables and data string variables. Data variables are equivalent to the registers of a register automaton, and data string variables store data words in order to generate the output. Streaming data string transducers are designed to model a class of list-processing programs.

In Chapter 2, we will introduce the above-mentioned models in more detail. Examples of models for words over infinite alphabets that are not discussed further in this work include pebble automata [20], data automata [4], class memory automata [3], fresh register automata [27], and set augmented finite automata [2].

We will then present some theoretical results for the examined models, and a prototype of a regex matcher based on register set automata. The regex matcher is based on the fact that register set automata can determinise a class of register automata. This allows our prototype to match inputs with amortized constant per-symbol complexity. As back-references are not expressible by finite automata, regex matchers must sometimes resort to back-tracking to match them. Back-tracking can, however, lead to the so-called catastrophic backtracking, which can cause massive slowdowns in matching the regex. Catastrophic back-tracking can be targeted by a ReDoS (regular expression denial of service) attack [28], where an attacker inputs a malicious text intended to cause serious slowdown in a regex matcher, making the targeted system unresponsive.

Chapter 2

Automata Models

This chapter will provide definitions and other useful information for the examined models. We will sometimes use $\cdot$ to denote an *ellipsis* (a value that can be ignored).

Alphabet and Data Domain. An *alphabet* Σ is a non-empty finite set whose elements are called *symbols* (sometimes called 'labels', 'tags', or similar in the context of data words). A *data domain* $\mathbb{D}$ is a countably infinite set of *data values* such that $\bot \notin \mathbb{D}$. Further in the text, we will often just use Σ and $\mathbb{D}$, assuming that they are an alphabet and a data domain, respectively, without explicitly stating it.

Data Words and Languages. Given a finite alphabet Σ and an infinite data domain $\mathbb{D}$, a *data word* w of length n over $\Sigma \times \mathbb{D}$ is a finite sequence $(a_1, d_1) \dots (a_n, d_n)$ where each $(a_i, d_i) \in \Sigma \times \mathbb{D}$. The *empty word* of length 0 is denoted as ε. A *data language* over $\Sigma \times \mathbb{D}$ is a set of words over $\Sigma \times \mathbb{D}$. For the sake of simplicity, data words and data languages will often be referred to as just *words* and *languages* throughout this work.

2.1 Register Automata

Register automata are an automata variant equipped with a set of registers, each storing a data value. The automaton can check the (non-)equality of the values stored in each register to the currently processed data value. We explore them mostly due to the fact that they are capable of representing a certain class of regular expressions with back-references.

A (non-deterministic) *register automaton* (NRA) over $\Sigma \times \mathbb{D}$ is a tuple $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$, where Q is a finite set of *states*, $\mathbf{R}$ is a finite set of *registers*, such that $\bot, \mathit{in} \notin \mathbf{R}$, $I \subseteq Q$ is the set of *initial states*, $F \subseteq Q$ is the set of *final states*, and $\Delta \subseteq Q \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \to (\mathbf{R} \cup \{\mathit{in}, \bot\})) \times Q$ is a *transition relation*, such that if $t = (q, a, g^{=}, g^{\neq}, \mathit{up}, s) \in \Delta$, then $g^{=} \cap g^{\neq} = \emptyset$. Semantically, this means that $\mathcal{A}$ can use transition t to move from q to s if the currently processed Σ-symbol is a and the currently processed $\mathbb{D}$-value (denoted in) is equal to the value stored in all registers in $g^{=}$ and no registers in $g^{\neq}$. The registers are then updated such that $r \leftarrow \mathit{up}(r)$ For better clarity, we will denote t as $q \xrightarrow{\boxed{a \mid g^{=}, g^{\neq}, \mathit{up}}} s$ and treat update mappings of the form $r \leftarrow r$ as implicit.

A *configuration* of $\mathcal{A}$ is a pair $c = (q, f) \in Q \times (R \to \mathbb{D} \cup \{\bot\})$, where q is the current state of $\mathcal{A}$ and f is the current register assignment. An *initial configuration* of $\mathcal{A}$ is a pair $c_{\mathit{init}} = (q_{\mathit{init}}, f_{\mathit{init}})$, where $q_{\mathit{init}} \in I$ and $f_{\mathit{init}} = \{r \mapsto \bot \mid r \in \mathbf{R}\}$. Let $c_1 = (q_1, f_1)$ and

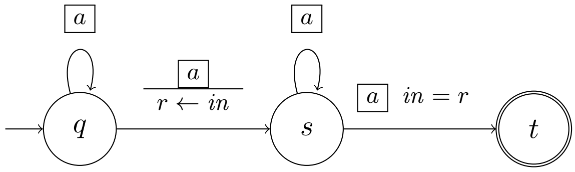


Figure 2.1: A non-deterministic RA, accepting the language of words whose last data value is in the word more that once.

$c_2 = (q_2, f_2)$ be two configurations of $\mathcal{A}$. We say that $\mathcal{A}$ can make a *step* from c_1 to c_2 over (a, d) using transition $t = q_1 \xrightarrow{\boxed{a \mid g^{=}, g^{\neq}, up}} q_2 \in \Delta$, denoted as $c_1 \vdash_t^{(a,d)} c_2$, iff

1. $\forall r \in g^{=} \colon d = f_1(r)$,
2. $\forall r \in g^{\neq} \colon d \neq f_1(r)$, and
3. $\forall r \in \mathbf{R} \colon f_2(r) = \begin{cases} f_1(r') & \text{if } up(r) = r' \in R, \\ d & \text{if } up(r) = in, \text{ and} \\ \bot & \text{if } up(r) = \bot. \end{cases}$

A *run* ρ of $\mathcal{A}$ over the word $w = (a_1, d_1) \dots (a_n, d_n)$ from a configuration c_0 is a sequence of configurations (c_i) and transitions (t_i) $\rho = c_0 t_1 c_1 t_2 \dots t_n c_n$, where for every $1 \leq i \leq n$ it holds that $c_{i-1} \vdash_{t_i}^{(a_i,d_i)} c_i$. We call ρ an *accepting run* if c_0 is an initial configuration and $c_n = (q, f)$, where $q \in F$. The language $L(\mathcal{A})$ accepted by $\mathcal{A}$ is the set of all words over which there exists an accepting run of $\mathcal{A}$.

We say that $\mathcal{A}$ is *deterministic* if $|I| \leq 1$ and if for every $q \in Q$ and every $a \in \Sigma$ it holds that for any two distinct transitions $q \xrightarrow{\boxed{a \mid g_1^{=}, g_1^{\neq}, up_1}} s_1, q \xrightarrow{\boxed{a \mid g_2^{=}, g_2^{\neq}, up_2}} s_2 \in \Delta$ we have $g_1^{=} \cap g_2^{\neq} \neq \emptyset$ or $g_1^{\neq} \cap g_2^{=} \neq \emptyset$. We will call deterministic register automata DRAs for short.

Example 1. Consider the language of words over $\Sigma = \{a\}$, whose last data value has appeared earlier in the word. An NRA accepting this language can be seen in Figure 2.1. Intuitively, the NRA waits in q, where it non-deterministically selects a value to store in r. It then waits in s for the last data symbol of the word, which it then compares to the one stored in r. It should be noted that this language is not expressible by a deterministic RA, and that its complement, a language of words whose last data value is unique in the word, is not expressible by any NRA.

Properties of RAs

We use C to describe both a class of automata and the class of languages accepted by the said class (e.g., RA for the register automata and the class of languages accepted by them). As mentioned above, RAs are capable of representing a certain class of regular expressions with back-references. The problem with doing so in a practical application (e.g., an efficient automata-based regex matcher) is that they are not determinisable in general.

Fact 1. [16, Remark 2] *DRA* $\subsetneq$ *NRA.*

Proof idea. Because NRA is not closed under complement and DRA is (see Fact 3 and Fact 4), there must be languages in NRA that are not in DRA (specifically the languages in NRA whose complement is not in NRA). See the language considered in Example 1, whose complement is not in NRA and thus the language itself cannot be in DRA. Trivially, all languages in DRA are also in NRA. □

Next, we take a look at the emptiness problem (the emptiness problem of an automaton is to determine whether the automaton's language is empty). We will mostly be concerned about the emptiness problems of the more complex automata models, but for the sake of completeness, we show the result for RA.

Fact 2. [7, Theorem 4.3 and Theorem 5.1] *The emptiness problem for NRA is PSPACE-complete.*

We follow with closure properties of RA under Boolean operations. One of the original ideas behind RA was to preserve closure properties on infinite alphabets. Except for closure under complement, this was successful.

Fact 3. [16, Theorem 3 and Remark 2] *NRA is closed under union and intersection. NRA is not closed under complement.*

Proof idea. Consider the language of words whose last data value has appeared earlier in the word from Example 1. The complement of this language, the language of words whose last data value is unique in the word, is inexpressible by NRA as it would require an unbounded number of registers to remember all the encountered data values. For union and intersection of two NRAs we can use the standard construction of an automaton that runs both NRAs in parallel and accepts if at least one of them accepts (for union) or accepts if both of them accept (for intersection). □

Fact 4. [16, Chapter 4] *DRA is closed under union, intersection, and complement.*

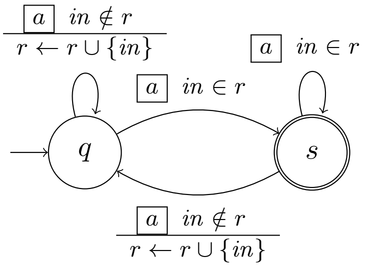
Proof idea. For union and intersection we can use the same standard construction as we did for NRAs. For complement we can simply change all final states to non-final and all non-final states to final states (note that the DRA needs to be complete). □

Closure under complement is a major point separating NRA from DRA. We will see a similar pattern in the more complex models.

2.2 Register Set Automata

Register set automata generalize register automata by allowing each register to hold a set of data values and checking the (non-)membership of the currently processed data value in them. The registers in RsAs will sometimes be called set-registers, in order to differentiate them from registers of RAs.

A *register set automaton* (RsA) over $\Sigma \times \mathbb{D}$ is a tuple $\mathcal{A}_S = (Q, \mathbf{R}, \Delta, I, F)$, where $Q, \mathbf{R}, I, F$ are the same as in an RA, and $\Delta \subseteq Q \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \to 2^{\mathbf{R} \cup \{in\}}) \times Q$ is a transition relation, such that if $t = (q_1, a, g^{\in}, g^{\notin}, up, q_2) \in \Delta$ (like in RAs, we will denote t as $q_1 \xrightarrow{a \mid g^{\in}, g^{\notin}, up} q_2$), then $g^{\in} \cap g^{\notin} = \emptyset$. The semantics of t is that $\mathcal{A}_S$ can move from state q_1 to state q_2 using t if the current Σ-symbol is a and the current $\mathbb{D}$-value (denoted as in) is in all the registers in $g^{\in}$ and in none of the registers in $g^{\notin}$. The registers are then updated so that $r \leftarrow \bigcup\{y \mid y \in up(r)\}$.



$\boxed{a}$ $in \notin r$
$r \leftarrow r \cup \{in\}$
$\boxed{a}$ $in \in r$
$\boxed{a}$ $in \in r$
q
s
$\boxed{a}$ $in \notin r$
$r \leftarrow r \cup \{in\}$

Figure 2.2: A deterministic RsA, accepting the language of words whose last data value is in the word more that once.

A *configuration* of $\mathcal{A}_S$ is a pair $c = (q, f) \in Q \times (\mathbf{R} \to 2^{\mathbb{D}})$, where q is the current state of $\mathcal{A}_S$ and f is the current register assignment. An *initial configuration* of $\mathcal{A}_S$ is a pair $c_{init} = (q_{init}, f_{init})$, where $q_{init} \in I$ and $f_{init} = \{r \mapsto \emptyset \mid r \in \mathbf{R}\}$. Let $c_1 = (q_1, f_1)$ and $c_2 = (q_2, f_2)$ be two configurations of $\mathcal{A}_S$. We say that $\mathcal{A}_S$ can make a step from c_1 to c_2 over (a, d) using transition $t = q_1 \xrightarrow{a \mid g^{\in}, g^{\notin}, up} q_2 \in \Delta$, denoted as $c_1 \vdash_t^{(a,d)} c_2$, iff

1. $\forall r \in g^{\in} \colon d \in f_1(r)$,
2. $\forall r \in g^{\notin} \colon d \notin f_1(r)$, and
3. $\forall r \in \mathbf{R} \colon f_2(r) = \bigcup \{f_1(r') \mid r' \in up(r), r' \in \mathbf{R}\} \cup \begin{cases} \{d\} & \text{if } in \in up(r) \text{, and} \\ \emptyset & \text{otherwise.} \end{cases}$

The run of $\mathcal{A}_S$ and the language accepted by $\mathcal{A}_S$ have the same definitions as for RAs.

We say that $\mathcal{A}_S$ is *deterministic* if $|I| \leq 1$ and if for all $q \in Q$ and all $a \in \Sigma$ it holds that for any two distinct transitions $q \xrightarrow{a \mid g_1^{\in}, g_1^{\notin}, up_1} s_1, q \xrightarrow{a \mid g_2^{\in}, g_2^{\notin}, up_2} s_2 \in \Delta$ we have $g_1^{\in} \cap g_2^{\notin} \neq \emptyset$ or $g_1^{\notin} \cap g_2^{\in} \neq \emptyset$.

Example 2. In this example, we recall the language from Example 1 (the language of all words whose last data symbol has appeared previously in the word). A deterministic RsA accepting this language is shown in Figure 2.2.

Properties of RsAs

We will look at some key properties of RsAs. First is the fact that RsAs generalize RAs.

Fact 5. [11, Fact 6] *For every $n \in \mathbb{N}$ and NRA$_n$, there exists an RsA$_n$ accepting the same language.*

Next, we turn to the emptiness problem for RsA. Although decidable, it is of staggering complexity.

Fact 6. [11, Theorem 7] *The emptiness problem for RsAs is Ackermann-complete.*

We follow with closure properties of RsAs. Observe that they are the same as the closure properties of RA. The proofs for the closure properties of RsA use the same ideas as those for RA.

Fact 7. [11, Theorem 9] *RsA is closed under union and intersection. RsA is not closed under complement.*

The deterministic variant also has the same closure properties as its deterministic RA counterpart.

Fact 8. [11, Theorem 11] *DRsA is closed under union, intersection, and complement.*

This gives rise to the fact that, as with RAs, the deterministic variant is strictly weaker in terms of expressive power.

Fact 9. [11, Theorem 13] *DRsA $\subsetneq$ RsA.*

The last fact is one of the most interesting properties of register set automata.

Fact 10. [11, Chapter 5] *A large class of NRAs can be determinised into DRsAs.*

This can be done using Algorithm 1 from [11]. The algorithm, at its core, is an extended version of the classical subset construction algorithm for determinising finite automata [21]. Each register of the RA has its copies created for each state it is active in, then the set registers of the DRsA track the sets of possible values that could be stored in the register. The algorithm will be described in more detail in Section 7.

2.3 RsAs with Removal

RsAs with Removal are an extension of the register set automata model, allowing the automaton to remove the currently processed value from a register, in addition to the capabilities of a normal RsA.

An *RsA with removal* (RsA^{rm}) over $\Sigma \times \mathbb{D}$ is a tuple $\mathcal{A}_R = (Q, \mathbf{R}, \Delta, I, F)$, where $Q, \mathbf{R}, I, F$ are the same as for RsAs and $\Delta \subseteq Q \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \to 2^{\mathbf{R} \cup \{in\}}) \times 2^{\mathbf{R}} \times Q$ is a transition relation such that if $t = (q_1, a, g^{\in}, g^{\notin}, up, rm, q_2) \in \Delta$ (we will denote t as $q_1 \xrightarrow{\boxed{a \mid g^{\in}, g^{\notin}, up, rm}} q_2$), then $g^{\in} \cap g^{\notin} = \emptyset$. The conditions under which a transition may be used are the same as in RsAs. The contents of registers are updated such that if $r \in rm$, then $r \leftarrow \bigcup\{x \mid x \in up(r)\} \setminus \{in\}$, else $r \leftarrow \bigcup\{x \mid x \in up(r)\}$.

Definitions for a *configuration* and an *initial configuration* are the same as for an RsA. Let $c_1 = (q_1, f_1)$ and $c_2 = (q_2, f_2)$ be two configurations of $\mathcal{A}_R$. We say that $\mathcal{A}_R$ can make a step from c_1 to c_2 over (a, d) using transition $t = q_1 \xrightarrow{\boxed{a \mid g^{\in}, g^{\notin}, up, rm}} q_2 \in \Delta$, denoted as $c_1 \vdash_t^{(a,d)} c_2$, iff

1. $\forall r \in g^{\in} \colon d \in f_1(r)$,
2. $\forall r \in g^{\notin} \colon d \notin f_1(r)$, and
3. $\forall r \in \mathbf{R} \colon f_2(r) = \bigcup\{f_1(r') \mid r' \in up(r), r' \in \mathbf{R}\} \cup x_1 \setminus x_2$, where $x_1 = \{d\}$, if $in \in up(r)$ and $x_1 = \emptyset$ otherwise, and $x_2 = \{d\}$, if $r \in rm$ and $x_2 = \emptyset$ otherwise.

The run on $\mathcal{A}_R$, the language accepted by $\mathcal{A}_R$, and determinism of $\mathcal{A}_R$ have the same definitions as for RsAs.

Example 3. We look at a language of words whose last two data values have appeared earlier in the string, but are different from one another. An RsA^{rm} accepting this language is shown in Figure 2.3. Intuitively, in the initial state q, the automaton collects all the values appearing in the first part of the string in r, then guesses when the second last symbol is read and removes the associated data value from r (after checking that the value has indeed appeared before), while moving to state s. Lastly, it checks that the last data value also appeared previously.

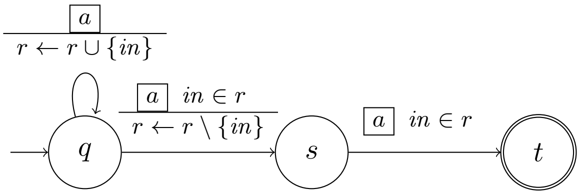


Figure 2.3: An RsA^{rm} accepting the language of words whose last two data values have appeared earlier in the string, but are different from one another

Properties of RsAs with Removal

RsA^{rm} have very similar properties to regular RsA. First, we note that they are a generalization of RsA.

Fact 11. *For every $n \in \mathbb{N}$ and RsA$_n$, there exists an RsA$_n^{rm}$ accepting the same language.*

Proof. Any RsA$_n$ can be converted to a RsA$_n^{rm}$ by simply adding an empty *rm* set to each transition. □

Although a generalization, they keep the same result for their emptiness problem.

Fact 12. [11, Theorem 37] *The emptiness problem for RsA^{rm}s is Ackermann-complete.*

The same is true for their closure properties.

Fact 13. *RsA^{rm} is closed under union and intersection. RsA^{rm} is not closed under complement.*

Proof idea. For union and intersection we can use the standard construction of running two RsA^{rm}s in parallel and either accepting if at least one of them accepts to get their union or accepting if both of them accept to get their intersection.

For non-closure under complement, we use the proof of Theorem 9 in [11] (closure properties of RsA, see Fact 7). The proof shows that if the language of words whose data values all have more than one occurrence were expressible by an RsA (the complement of this language, the language of words with at least one unique data value is expressible by RsA), then RsA could be used to encode the accepting runs of a Minsky machine. Because RsA emptiness is decidable, and Minsky machine emptiness is not [19], this is a contradiction.

The same argument also applies to RsA^{rm}, as their emptiness is also decidable and every language expressible by RsA is expressible by RsA^{rm}. □

Following the pattern started by RAs, the deterministic variant of RsAs with removal is also strictly weaker and also closed under complement.

Fact 14. *DRsA^{rm} is closed under union, intersection, and complement.*

Proof idea. For union and intersection, we can use the same construction as we used for the non-deterministic variant. For complement, we can simply swap final and non-final states (note that the DRsA^{rm} needs to be complete). □

Fact 15. *DRsA^{rm} $\subsetneq$ RsA^{rm}.*

Proof idea. We know that DRsA^{rm} are closed under complement and RsA^{rm} are not and that every DRsA^{rm} is also an RsA^{rm}. From this we can conclude that there must be languages that are expressible by a RsA^{rm} and not expressible a DRsA^{rm} as there would be a contradiction otherwise. □

2.4 History Register Automata

History register automata, presented in [10], are similar to the RsA^{rm} model in the fact that they also allow sets of values to be stored in registers. The key difference is the assignment of values to registers. In a history register automaton, one can only change the contents of registers by adding or removing the currently processed data value. Note that the under the original definition, history register automata are slightly different from how they are presented here. Originally, they do not run on data words (and thus don't have Σ-labels on transitions), they have initial register assignments, and they have a set of RA-like registers (simply referred to as 'registers' in the original work) along with the set-registers (referred to as 'histories'). This change was made to align them better with RsA^{rm}s (one could just as easily modify RsA^{rm}s to match them up better with the original history register automata definition).

A *history register automaton* (HRA) over $\mathbb{D} \times \Sigma$ is a tuple $\mathcal{A}_H = (Q, \mathbf{R}, \Delta, I, F)$, where $Q, \mathbf{R}, I, F$ is the same as in RsAs, and $\Delta \subseteq Q \times \Sigma \times ((2^{\mathbf{R}} \times 2^{\mathbf{R}}) \cup 2^{\mathbf{R}}) \times Q$. In this model, there are two types of transitions:

1. Updating transitions $t_{up} = q \xrightarrow{a \mid R_g, R_{up}} q'$. $\mathcal{A}_H$ can use such a t_{up} if the current state is q, the current Σ-symbol is a and in is in all the registers in R_g and in none of the registers in $\mathbf{R} \setminus R_g$. The contents of the registers are then updated so that in is in all the registers in R_{up} and in none of the registers in $\mathbf{R} \setminus R_{up}$.

2. Resetting transitions, $t_{res} = q \xrightarrow{R_{clr}} q'$. $\mathcal{A}_H$ can use such a t_{res} if the q is the current state. They do not consume any input symbols (they are ε-transitions). The register contents are then updated so that all the registers in R_{clr} are emptied (the other registers are left as they are).

Definitions for a *configuration* and an *initial configuration* are the same as for an RsA. Let $c_1 = (q_1, f_1), c_2 = (q_2, f_2)$ be two configurations of $\mathcal{A}_H$. We say that $\mathcal{A}_H$ can make a step from c_1 to c_2 over (a, d) using an update transition $t_{up} = q_1 \xrightarrow{a \mid R_g, R_{up}} q_2$, denoted as $c_1 \vdash_{t_{up}}^{(a,d)} c_2$, iff

1. $\forall r \in R_g \colon d \in f_1(r)$,

2. $\forall r \in \mathbf{R} \setminus R_g \colon d \notin f_1(r)$,

3. $\forall r \in R_{up} \colon f_2(r) = f_1(r) \cup \{d\}$, and

4. $\forall r \in \mathbf{R} \setminus R_{up} \colon f_2(r) = f_1(r) \setminus \{d\}$.

$\mathcal{A}_H$ can also make a step from c_1 to c_2 using a reset transition $t_{res} = q_1 \xrightarrow{R_{clr}} q_2$, consuming no input, denoted as $c_1 \vdash_{t_{res}}^{\varepsilon} c_2$, iff

1. $\forall r \in R_{clr} \colon f_2(r) = \emptyset$, and

2. $\forall r \in \mathbf{R} \setminus R_{clr} \colon f_2(r) = f_1(r)$.

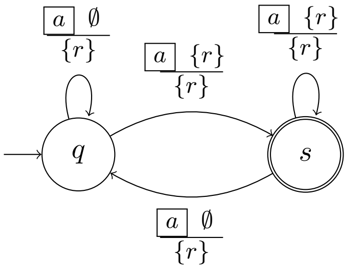


Figure 2.4: A deterministic HRA, accepting the language of words whose last data value is in the word more that once.

The run on $\mathcal{A}_H$ and the language accepted by $\mathcal{A}_H$ have the same definitions as for RsAs. We say that $\mathcal{A}_H$ is *deterministic* if for all states $q_1 \in Q$ it holds that either

1. there is only one transition originating in q_1, or
2. there are no reset transitions originating in q_1, and for all $a \in \Sigma, q_2 \in Q$ it holds that there are no distinct transitions $q_1 \xrightarrow{a \mid R_g^1, R_{up}^1} q_2, q_1 \xrightarrow{a \mid R_g^2, R_{up}^2} q_2 \in \Delta$ such that $R_g^1 = R_g^2$.

Example 4. We look to the language (of words whose last data value was present earlier in the word) from Example 1 and Example 2 one more time. See the DHRA accepting it in Figure 2.4. Observe that the automaton is the same as the DRsA in Example 2, except that the transition labels use different notation. The set at the top of the label is the guard set, specifying which registers contain *in*, and the set at the bottom of the label is the update set, specifying in which registers should *in* be after the transition.

Properties of HRAs

Although defined differently and created for different purposes, HRAs have very similar properties to RsAs (and we will directly compare them later). First we note that HRAs also generalize RAs.

Fact 16. [10, Definition 2.4 and Proposition 4.1] *For every $n \in \mathbb{N}$ and NRA$_n$, there exists an HRA$_n$ accepting the same language.*

HRA emptiness problem has the same result as its RsA counterpart.

Fact 17. [10, Proposition 5.4] *The emptiness problem for HRAs is Ackermann-complete.*

HRAs also have the same closure properties as RsAs.

Fact 18. [10, Proposition 3.2 and Lemma 3.3] *HRA is closed under union and intersection. HRA is not closed under complement.*

And as with all previous models, their deterministic variant is also strictly weaker and closed under complement.

Fact 19. *DHRA is closed under union, intersection, and complement.*

Proof idea. For union and intersection we can use the standard construction of running two DHRAs in parallel and either accepting if at least of them accepts to get their union or accepting if both of them accept to get their intersection. For complement we can simply swap final and non-final states (note that the DHRA needs to be complete). □

Fact 20. *DHRA* $\subsetneq$ *HRA.*

Proof idea. DHRA is closed under complement, while HRA is not and every DHRA is also a HRA. From this we can conclude that there must be languages in HRA that are not in DHRA, because otherwise we would have a contradiction. □

2.5 Streaming Data String Transducers

Streaming data string transducers, proposed in [1], are a transducer model designed as a model for analyzing programs that access and modify lists of data items in a single pass. They are equipped with data variables, which are essentially registers from RAs, each storing a data value that can be compared to the input on transition guards, and data string variables, storing data strings for the purpose of generating the output. Data string variables can be updated by a concatenation of data string variables and the data values stored in data variables paired with symbols of the output alphabet, with the restriction that each data string variable can only appear once in a right-hand-side expression on a transition.

Because streaming data string transducers use an ordering on the data domain, we define $\mathbb{D}^<$ as a countably infinite set of data values with a strict total order $<$ over it. A *(deterministic) data transduction* over $\mathbb{D}^<$ from an input alphabet Σ to an output alphabet Γ is a partial function F from $(\Sigma \times \mathbb{D}^<)^*$ to $(\Gamma \times \mathbb{D}^<)^*$.

A *(deterministic) streaming string transducer* (SDST) $\mathcal{S}$ over $\mathbb{D}^<$ from an input alphabet Σ to an output alphabet Γ is a tuple $(Q, q_i, V, X, O, \Delta)$, where Q is a finite set of states, $q_i \in Q$ is an initial state, V is a finite set of data variables including $\mathit{in} \in V$, a special variable referring to the data value of the current input symbol, X is a finite set of data string variables, O is a partial output function from Q to $((\Gamma \times V) \cup X)^*$ and Δ is a finite set of transitions of the form $q \xrightarrow{a \mid \varphi, \mathit{up}} q'$, where $q \in Q$ is a source state, $a \in \Sigma$ is the input Σ-symbol, φ is a Boolean formula over atomic constraints of the form $v < \mathit{in}$ and $\mathit{in} < v$ with $v \in V$, q' is a target state and up is an update mapping V to V and X to $((\Gamma \times V) \cup X)^*$. Furthermore, it is required that (i) for all $q \in Q, x \in X$ there is at most one occurrence of x in $O(q)$, (ii) for each $q \xrightarrow{a \mid \varphi, \mathit{up}} q' \in \Delta, x \in X$ there is at most one occurrence of x in the set of strings $\{\mathit{up}(y) \mid y \in X\}$, and (iii) for each $q \xrightarrow{a \mid \varphi, \mathit{up}} q', q \xrightarrow{a \mid \varphi', \mathit{up}'} q'' \in \Delta$ the guards φ and φ' are mutually exclusive ($\varphi \wedge \varphi'$ is unsatisfiable).

A *configuration* of $\mathcal{S}$ is a pair $c = (q, f) \subseteq Q \times ((V \to (\mathbb{D}^< \cup \{\bot\})) \cup (X \to (\Gamma \times \mathbb{D}^<)^*))$, where q is the current state of $\mathcal{S}$ and f is the current variable assignment. The initial configuration of $\mathcal{S}$ is a pair $c_{\mathit{init}} = (q_i, f_i)$, where $f_i = \{v \mapsto \bot \mid v \in V\} \cup \{x \mapsto \varepsilon \mid x \in X\}$. For any variable assignment f we also define $f^{\mathit{eval}} \colon ((\Gamma \times V) \times X)^* \to (\Gamma \times \mathbb{D}^<)^*$, which evaluates a right-hand side of a data string variable update. It is defined for a word $y = y_1 \ldots y_n$ as $f^{\mathit{eval}}(y) = \mathit{eval}(y_1) \ldots \mathit{eval}(y_n)$, where $\mathit{eval}(y_i) = (a, f(v))$ if $y_i = (a, v), a \in \Gamma, v \in V$ and $\mathit{eval}(y_i) = f(y_i)$, if $y_i \in X$.

Let $c_1 = (q_1, f_1)$ and $c_2 = (q_2, f_2)$ be two configurations of $\mathcal{S}$. We say that $\mathcal{S}$ can make a step from c_1 to c_2 over (a, d) using transition $t = q_1 \xrightarrow{a \mid \varphi, \mathit{up}} q_2$, denoted as $c_1 \vdash_t^{(a,d)} c_2$ iff $f_1' = f_1[\mathit{in} \mapsto d]$ satisfies φ, $\forall v \in V \colon f_2(v) = f_1'(\mathit{up}(v))$, and $\forall x \in X \colon f_2(x) = f_1'^{\mathit{eval}}(\mathit{up}(x))$.

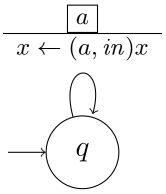


Figure 2.5: A SDST performing the transduction $F_{reverse}$, reversing any input data word

A run ρ of $\mathcal{S}$ over the word $w = (a_1, d_1) \dots (a_n, d_n)$ from a configuration c_0 is a sequence of configurations and transitions $\rho = c_0 t_0 c_1 t_1 \dots t_n c_n$, where for all $1 \leq i \leq n$ it holds that $c_{i-1} \vdash_{t_i}^{(a_i, d_i)} c_i$. We can denote ρ as $c_0 \vdash^w c_n$. The transduction $[\![\mathcal{S}]\!]$ of $\mathcal{S}$ is defined for an input word w as $[\![\mathcal{S}]\!](w) = f^{eval}(O(q))$, if there exists a run $c_{init} \vdash^w (q, f)$ and $O(q)$ is defined, otherwise $[\![\mathcal{S}]\!](w)$ is undefined. The *functional equivalence problem* for SDSTs is, given two SDSTs $\mathcal{S}_1$ and $\mathcal{S}_2$, whether $[\![\mathcal{S}_1]\!] = [\![\mathcal{S}_2]\!]$.

Example 5. Consider the transduction $F_{reverse}$ over $\mathbb{D}^<$ from $\Sigma = \{a\}$ to $\Gamma = \{a\}$ that reverses any input data word. See the SDST performing $F_{reverse}$ in Figure 2.5. The transducer has one state q and one data string variable x. At each step it puts the current symbol at the beginning of x. The output function is defined in q to be x.

Properties of SDSTs

For the purposes of program verification, a class of single-pass list-processing imperative programs and a class of single-pass list-processing functional programs were defined in [1]. These programs can be represented by SDSTs.

Fact 21. [1, Proposition 6] *Given a single-pass list-processing program P, one can effectively construct an SDST $\mathcal{S}$ such that $[\![P]\!] = [\![\mathcal{S}]\!]$.*

Fact 22. [1, Proposition 8] *Given a single-pass list-processing function f, one can effectively construct an SDST $\mathcal{S}$ such that $[\![f]\!] = [\![\mathcal{S}]\!]$.*

In fact, the reverse is also true and it is possible to construct list-processing imperative and functional programs from a given SDST.

Fact 23. [1, Proposition 7] *Given an SDST $\mathcal{S}$, one can effectively construct a single-pass list-processing program P such that $[\![\mathcal{S}]\!] = [\![P]\!]$.*

Fact 24. [1, Proposition 9] *Given an SDST $\mathcal{S}$, one can effectively construct a single-pass list-processing function f such that $[\![\mathcal{S}]\!] = [\![f]\!]$.*

Next we move on to one of the main results of [1], which is that functional equivalence of SDSTs is decidable. This allows, e.g., to to check whether a functional list-processing program is semantically equivalent to an imperative list-processing program.

Fact 25. [1, Theorem 12] *The SDTS functional equivalence problem is in PSPACE.*

Chapter 3

Vector Addition Systems with States

Vector addition systems with states are a model used (not only) for description of distributed systems. We will use them to reason about some decidability problems of the studied automata models. Vector addition systems with states have a finite number of counters over natural numbers, which are updated by integer vectors on the transitions. A transition can only be taken if no counter is lowered below zero by adding the transition's vector.

We use $\mathbb{N}$ and $\mathbb{Z}$ to denote the sets of natural numbers and integers, respectively. We use $\vec{0}_d$ to denote the zero vector of dimension d, $v[i]$ to denote the value of i-th dimension of the vector, $v[i \mapsto n]$ to denote the vector v with the i-th dimension set to n, and $v_1 \leq v_2$ to denote that $v_1[i] \leq v_2[i]$ for all i.

A *vector addition system with states* (VASS) of *dimension* d is a tuple $\mathcal{V} = (Q, \Delta, q_i)$, where Q is a finite set of *control states*, $q_i \in Q$ is the *initial control state* and $\Delta \subseteq Q \times \mathbb{Z}^d \times Q$ is a *transition relation.* The semantics of a transition $t = q_1 \xrightarrow{\boxed{(a_1, \ldots, a_d)}} q_2 \in \Delta$ is that $\mathcal{V}$ can move from q_1 to q_2, if adding each a_i to the i-th counter does not lower the counter's value below 0. Each i-th counter is the updated by adding the value of a_i to it.

A *configuration* of $\mathcal{V}$ is a pair $c = (q, v) \in Q \times (\mathbb{N}^d)$, where q is the current control state and v is a vector (of dimension d) of the current counter values. The *initial configuration* of $\mathcal{V}$ is $c_{init} = (q_i, \vec{0}_d)$. Let $c_1 = (q_1, v_1), c_2 = (q_2, v_2)$ be two configurations of $\mathcal{V}$. We say that $\mathcal{V}$ can move from c_1 to c_2 using t, denoted $c_1 \vdash_t c_2$ iff $\vec{0} \leq v_1 + v = v_2$.

The *reachability problem* for a VASS is defined as whether a configuration c_{dst} is reachable from another configuration c_{src}. More precisely, c_{dst} is reachable from c_{src} if there exists some sequence of configurations and transitions $c_0 t_1 c_1 t_2 \ldots t_n c_n$, such that $c_{src} = c_0, c_{dst} = c_n$, and for all $1 \leq i \leq n$ it holds that $c_{i-1} \vdash_{t_i} c_i$

The *coverability problem* for a VASS is defined as whether a configuration $c_{dst} = (q_{dst}, v_{dst})$ is coverable from another configuration c_{src}, that is, whether some configuration $c' = (q_{dst}, v')$, where $v_{dst} \leq v'$, is reachable from c_{src}.

We can use the coverability problem to define the *control state reachability* problem, the problem asking whether a control state q is reachable from a configuration c_{src}. This is equivalent to asking whether the configuration $(q, \vec{0}_d)$ is coverable from c_{src}.

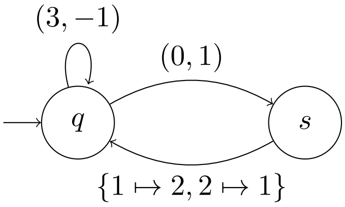


Figure 3.1: An example T-VASS

3.1 Extensions

VASSes have many different extensions, but here we will only look at adding reset or transfer transitions to our existing definition of a VASS. We will be calling VASSes equipped with reset and transfer transition reset-VASSes (R-VASS) and transfer-VASSes (T-VASS), respectively.

A *reset* transition is a transition of the form $t^r = q_1 \xrightarrow{\boxed{\mathit{rst}}} q_2$, where $\mathit{rst} \subseteq \{1, \ldots, d\}$ is a set, specifying which counters should be reset. Let $c_1 = (q_1, v_1), c_2 = (q_2, v_2)$ be two configurations of a R-VASS $\mathcal{V} = (Q, \Delta, q_i)$. $\mathcal{V}$ can use t^r (provided $t^r \in \Delta$) to move from c_1 to c_2, denoted $c_1 \vdash_{t^r} c_2$ iff for all $i \in \{1, \ldots, d\}$ it holds that if $i \in \mathit{rst}$ then $v_2[i] = 0$ and $v_1[i] = v_2[i]$ otherwise.

A *transfer* transition a transition of the form $t^t = q_1 \xrightarrow{\boxed{\mathit{tr}}} q_2$, where tr is a total transfer function from $\{1, \ldots, d\}$ to $\{1, \ldots, d\}$. Let $c_1 = (q_1, v_1), c_2 = (q_2, v_2)$ be two configurations of a T-VASS $\mathcal{V} = (Q, \Delta, q_i)$. $\mathcal{V}$ can use t^t (provided $t^t \in \Delta$) to move from c_1 to c_2, denoted $c_1 \vdash_{t^t} c_2$ iff for all $i \in \{1, \ldots, d\}$ it holds that $v_2[i] = \sum_{j \in \{x \mid \mathit{tr}(x) = i\}} v_1[j]$.

The reachability, coverability and control state reachability problems are defined the same way for T-VASSes and R-VASSes as for regular VASSes.

Example 6. Figure 3.1 shows an example of a 2-dimensional T-VASS. The transition from q to q increases the value of the first counter by 3, while decreasing the value of the second counter by 1 (and thus the transition cannot be taken if the value of the second counter is zero). The transition from q to s increases the value of the second counter by one, and the transition from s to q is a transfer transition that swaps the values of the two counters.

3.2 Grzegorczyk Hierarchy

The Grzegorczyk hierarchy $(\mathbf{F}_k)_{k<\omega}$ is a hierarchy of classes of primitive-recursive functions $f \colon \mathbb{N} \to \mathbb{N}$. We will briefly introduce it here, as we use it to express some complexity results (the definitions used here are as specified in [23, Section 2.1.3]). Each class $\mathbf{F}_k$ is defined using its fast-growing function $F_k \colon \mathbb{N} \to \mathbb{N}$ as $\mathbf{F}_k = \{f \mid \exists i \colon f \text{ is computed in time/space } \leq F_k^i\}$ (the difference between time and space complexity is irrelevant as F_2 is already of exponential growth). The fast-growing functions are defined inductively as $F_0 = x + 1$ and $F_{k+1} = F_k^{x+1}$, with f^i defined as the function obtained by composing f with itself i times.

For example, $\mathbf{F}_0 = \mathbf{F}_1$ is the class of linear functions, $\mathbf{F}_2$ already corresponds to exponential complexity, $\mathbf{F}_3$ to exponent tower complexity and so on. An important thing to note is that for $k \geq 1$ the hierarchy is strict: $\mathbf{F}_k \subsetneq \mathbf{F}_{k+1}$. The union of all $\mathbf{F}_k$ (for finite k) is equal to the set of primitive-recursive functions. The hierarchy can be extended to ordinals, with $\mathbf{F}_\omega$ corresponding to Ackermannian complexity (defined by the fast-growing function $F_\omega(x) = F_x(x)$).

3.3 Well Quasi Orderings

A *quasi ordering* is a reflexive and transitive relation $\preceq$ over a set A. A *well quasi ordering* (wqo) $\preceq$ over a set A is a quasi ordering such that in every infinite sequence $x_1, x_2, \dots$ over A there exists i, j such that $i < j$ and $x_i \preceq x_j$ (i.e., there is an *increasing pair*).

A *bad sequence* over a wqo $(A, \preceq)$ is a sequence $x_1, x_2, \dots$ that contains no increasing pairs, i.e., $\forall i < j \colon x_i \not\preceq x_j$. By the definition of a wqo, all bad sequences are necessarily finite. Sequences that do contain an increasing pair are called *good sequences*.

Bad sequences can, in general, be of arbitrary length [23]. In order to bound bad sequences $s = x_1, x_2, \dots$ over a wqo $(A, \preceq)$, one needs a *norm* and a *control function*. A norm is some function $|\cdot|_A \colon A \to \mathbb{N}$ representing the size of elements of A. A control function is a function $g \colon \mathbb{N} \to \mathbb{N}$ that bounds the growth of the sequence from one element to the next. Thus s is controlled by g if for all i it holds that $|x_{i+1}|_A \leq g(|x_i|_A)$.

Lemma 1 (Length Function Theorem)**.** [23, Theorem 2.8] *Let $d \geq 0$ and g be a control function bounded by some function in $\mathbf{F}_\gamma$ for some $\gamma \geq 1$. Then the length of g-controlled bad sequences of a d-dimensional VASS's configurations is bounded by a function in $\mathbf{F}_{\gamma+d}$.*

Chapter 4

Relating RsAs and HRAs

In this chapter, we will examine the relationship of the RsA^{rm} and HRA models. Namely, we will compare their respective expressive powers (for both their non-deterministic and deterministic variants). The first observation one can make is that HRAs are convertible to RsA^{rm}s.

Proposition 1. *HRA* $\subseteq$ *RsA*^{rm}

Proof. We show that any HRA $\mathcal{A}_H = (Q_H, \mathbf{R}_H, \Delta_H, I_H, F_H)$ can be converted to an RsA^{rm} $\mathcal{A}_R = (Q_R, \mathbf{R}_R, \Delta_R, I_R, F_R)$ such that $L(\mathcal{A}_H) = L(\mathcal{A}_R)$. We keep the states, the initial states, and registers the same, i.e., $Q_H = Q_R, I_H = I_R, \mathbf{R}_H = \mathbf{R}_R$. We convert all update transitions of $\mathcal{A}_H$ $q_1 \xrightarrow{\boxed{a \mid R_g, R_{up}}} q_2 \in \Delta_H$ to $q_1 \xrightarrow{\boxed{a \mid g^{\in}, g^{\notin}, up, rm}} q_2 \in \Delta_R$, where $g^{\in} = R_g, g^{\notin} = \mathbf{R}_H \setminus R_g, rm = R_g \setminus R_{up}$, and for all $r \in \mathbf{R}_H$ if $r \in R_{up}$, then $up(r) = \{r, in\}$, otherwise $up(r) = \{r\}$.

To deal with reset transitions, we find will sequences of transitions in $\mathcal{A}_H$, such that the last transition is an update transition, and all the previous transitions are reset transitions. We also make sure the transitions form a path in $\mathcal{A}_H$, and that no transition is in the sequence more than once. We then create a transition in $\mathcal{A}_R$ that executes the transition sequence in one step.

We do so by first finding all sequences of transitions $t_1 = q_1 \xrightarrow{\boxed{R^1_{clr}}} q_2, t_2 = q_2 \xrightarrow{\boxed{R^2_{clr}}} q_3, \ldots, t_n = q_n \xrightarrow{\boxed{a \mid R_g, R_{up}}} q_{n+1}$, where $\forall 1 \leq i \leq n \colon t_i \in \Delta_H$, and $\forall 1 \leq i < j \leq n \colon t_i \neq t_j$. For each such sequence, where $R_g \cap \bigcup_{i=1}^{n-1} R^i_{clr} = \emptyset$, we add a transition $t = q_1 \xrightarrow{\boxed{a \mid g^{\in}, g^{\notin}, up, rm}} q_n$ to Δ_R, where $g^{\in} = R_g, g^{\notin} = \mathbf{R} \setminus R_g, rm = R_g \setminus R_{up}$, and for all $r \in \mathbf{R}_H$ if $r \in \bigcup_{i=1}^{n-1} R^i_{clr}$, then $up(r) = y$, otherwise $up(r) = \{r\} \cup y$, where $y = \{in\}$ if $r \in R_{up}$ and $y = \emptyset$ otherwise.

The final states F_R of $\mathcal{A}_R$ will then be the states F_H along with states that can reach any state $q_f \in F_H$ by a sequence of reset transitions. □

Notice that the same can be done for deterministic HRAs without introducing any non-determinism, which will be useful when relating the models' deterministic variants.

Corollary 1. *DHRA* $\subseteq$ *DRsA*^{rm}

The other direction of Proposition 1 was left as an open problem.

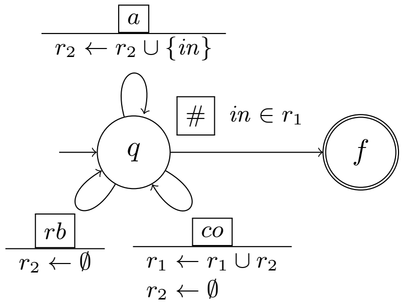

a
$r_2 \leftarrow r_2 \cup \{in\}$
#
$in \in r_1$
q
f
rb
$r_2 \leftarrow \emptyset$
co
$r_1 \leftarrow r_1 \cup r_2$
$r_2 \leftarrow \emptyset$

Figure 4.1: A DRsA^{rm} $\mathcal{A}$ accepting the language $L_{transact}$

4.1 Relating DRsAs and DHRAs

To compare the expressive power of DRsA^{rm}s and DHRAs we will use the language $L_{transact}$ of the DRsA^{rm} $\mathcal{A}$ shown in Figure 4.1. $L_{transact}$ is a language over the alphabet $\Sigma = \{a, co, rb, \#\}$ and the data domain $\mathbb{D}$. The semantics of $L_{transact}$ is as follows — data values of the symbol a are *committed* if the next non-a symbol is co or *rolled back* if the next non-a symbol is rb. Words are only part of $L_{transact}$ if their last symbol is # and its data value was committed earlier in the word (# may only appear as the last symbol of a word).

Lemma 2. *DRsA^{rm} $\not\subseteq$ DHRA*

Proof. Let us take the language $L_{transact}$ as defined above and assume there is a deterministic HRA $\mathcal{A}_H$ with n registers accepting it. Now let us look at the word $w = (a, d_1)(co, \cdot)(a, d_2)(co, \cdot) \dots (a, d_n)(co, \cdot)(a, d_{n+1})(co, \cdot)(\#, d)$, where $\forall i, j \colon i \neq j \implies d_i \neq d_j$. The word w belongs in $L_{transact}$ iff $\exists i \in \{1, \dots, n+1\} \colon d = d_i$.

Because $\mathcal{A}_H$ is deterministic (therefore there is only one possible configuration at any given point in the input word), $\mathcal{A}_H$ must store every data value of a in w until a # appears. If the data value of the k-th a in w were not stored in some register of $\mathcal{A}_H$ (or the register was emptied before # was reached), there is no way for $\mathcal{A}_H$ to distinguish between w where $d = d_k$, which belongs in $L_{transact}$, and w where d is a value not equal to any other data value in w, which does not belong in $L_{transact}$.

Using the pigeonhole principle we can then deduce that at least two data values of a in w must be stored in the same register. Let $l, m \in \mathbb{N}$, such that $l < m$ and d_l and d_m are the first two data values of a stored in one register. We then look at the word w' that is the same as w, except the pair $(co, \cdot)$ following the m-th a has been replaced with the pair $(rb, \cdot)$. As $\mathcal{A}_H$ is deterministic and words w and w' are the same until after (a, d_m) appears, we know that when reading w', d_l and d_m will be stored in the same register, but d_l was committed, whereas d_m was rolled back. This means that $\mathcal{A}_H$ loses the distinction between d_l and d_m and would either accept w' where $d = d_m$ or reject w' where $d = d_l$. This is a contradiction with the assumption that $\mathcal{A}_H$ accepts $L_{transact}$ and we can conclude that no DHRA can accept $L_{transact}$. □

Thus, we have shown that the deterministic variant of RsA^{rm} is strictly more expressive than the deterministic variant of HRA.

Proposition 2. *DHRA $\subsetneq$ DRsA^{rm}*

Proof. Follows from Corollary 1 and Lemma 2. □

Chapter 5

Parametrization of RsA Emptiness Complexity

Although the emptiness of RsA is known to be Ackermann-complete, in this chapter, we will parametrize the complexity of the problem based on the number of registers. We will do so for both standard RsAs and RsAs with removal. We start with RsAs with only one register.

5.1 Emptiness of RsA$_1$

We will obtain the complexity of RsA$_1$ emptiness by reducing it to and from non-deterministic finite automata (NFA) emptiness. We start with the upper bound.

Lemma 3. *The emptiness problem for RsA$_1$ is in NL.*

Proof. We reduce RsA$_1$ emptiness to NFA emptiness, which is NL-complete [15]. Given an RsA$_1$ $\mathcal{A} = (Q, \{r\}, \Delta, I, F)$ we construct an NFA $\mathcal{A}' = (Q', \Delta', I', F')$ whose language is empty iff the language of $\mathcal{A}$ is empty. Intuitively, we do this by tracking whether the register in $\mathcal{A}$ is empty in each state, and only including transitions that we can traverse using that information. Formally:

$$\begin{aligned}
Q' =& Q \times \{0, \omega\} \\
I' =& \{(q, 0) \mid q \in I\} \\
F' =& F \times \{0, \omega\} \\
\Delta' =& \{(q,0) \xrightarrow{\boxed{a}} (s,0) \mid q \xrightarrow{\boxed{a \mid \emptyset, g^{\notin}, \{r \leftarrow y\}}} s \in \Delta, \text{ where } in \notin y\} \\
&\cup \{(q,0) \xrightarrow{\boxed{a}} (s,\omega) \mid q \xrightarrow{\boxed{a \mid \emptyset, g^{\notin}, \{r \leftarrow y\}}} s \in \Delta, \text{ where } in \in y\} \\
&\cup \{(q,\omega) \xrightarrow{\boxed{a}} (s,0) \mid q \xrightarrow{\boxed{a \mid g^{\in}, g^{\notin}, \{r \leftarrow \emptyset\}}} s \in \Delta\} \\
&\cup \{(q,\omega) \xrightarrow{\boxed{a}} (s,\omega) \mid q \xrightarrow{\boxed{a \mid g^{\in}, g^{\notin}, \{r \leftarrow y\}}} s \in \Delta, \text{ where } y \neq \emptyset\}.
\end{aligned}$$

This is a log space reduction as the transitions are processed one at a time. □

Lemma 4. *The emptiness problem for RsA$_1$ is NL-hard.*

Proof. NFA emptiness, which is NL-complete, is a special case of RsA$_1$ emptiness (if the register is never assigned or tested, the RsA$_1$ can be treated as an NFA). □

Proposition 3. *The emptiness problem for RsA_1 is NL-complete.*

Proof. Follows from Lemma 3 and Lemma 4. □

5.2 Emptiness of RsA_1^{rm}

We obtain the upper bound for RsA_1^{rm} emptiness by reducing it to control state reachability of a a one-dimensional VASS.

Lemma 5. [10, Lemma 6.7] *Control state reachability for one dimensional R-VASSs is in NL, provided that non-reset transitions increase and decrease the counter by at most one.*

Lemma 6. *The emptiness problem for RsA_1^{rm} is in NL.*

Proof. We perform a log space reduction of RsA_1^{rm} emptiness to one dimensional R-VASS control state reachability. The idea is to use the counter to keep track of the number of values stored in the register. First we create the R-VASS $\mathcal{V} = (Q_\mathcal{V}, T, i)$ from the RsA_1^{rm} $\mathcal{A} = (Q_\mathcal{A}, \{r\}, \Delta, I, F)$. $Q_\mathcal{V}$ is defined such that $Q_\mathcal{A} \subseteq Q_\mathcal{V}$. For each transition $t = q \xrightarrow{\boxed{a \mid g^\in, g^\notin, up, rm}} s$ in $\mathcal{A}$ there are four transitions t_1, t_2, t_3, t_4 (in this order) in the R-VASS forming a path between q and s. If r is in neither $g^\in$ nor $g^\notin$, we consider t as two transitions covering both cases. Transitions t_1 and t_2 respectively check if the guards hold and fix the counter's value if needed. Transition t_3 resets the counter if $r \notin up(in)$, otherwise it adds or subtracts one from the counter based on whether in is added or removed. Transition t_4 then adds one to the counter if t_3 was a reset transition and in is added to r. The transitions t_1, t_2, t_3, t_4 are computed in the following way:

- $t_1\colon \begin{cases} q \xrightarrow{\boxed{-1}} q_1^t & \text{if } r \in g^\in \text{ and} \\ q \xrightarrow{\boxed{0}} q_1^t & \text{otherwise,} \end{cases}$
- $t_2\colon \begin{cases} q_1^t \xrightarrow{\boxed{+1}} q_2^t & \text{if } r \in g^\in \text{ and} \\ q_1^t \xrightarrow{\boxed{0}} q_2^t & \text{otherwise,} \end{cases}$
- $t_3\colon \begin{cases} q_2^t \xrightarrow{\boxed{\texttt{reset}(r)}} q_3^t & \text{if } r \notin up(r), \\ q_2^t \xrightarrow{\boxed{+1}} q_3^t & \text{if } r \in up(r) \wedge r \in g^\notin \wedge in \in up(r), \\ q_2^t \xrightarrow{\boxed{-1}} q_3^t & \text{if } r \in up(r) \wedge r \in g^\in \wedge r \in rm, \text{ and} \\ q_2^t \xrightarrow{\boxed{0}} q_3^t & \text{otherwise,} \end{cases}$
- $t_4\colon \begin{cases} q_3^t \xrightarrow{\boxed{+1}} s & \text{if } r \notin up(r) \wedge in \in up(r) \text{ and} \\ q_3^t \xrightarrow{\boxed{0}} s & \text{otherwise,} \end{cases}$

where $q_1^t, q_2^t, q_3^t \notin Q_\mathcal{A}$ are unique to each transition t being processed. Then we add the states *init*, *end* to $\mathcal{V}$, where *init* only has transitions into every initial state od $\mathcal{A}$ and *end* only has transitions from every final state in $\mathcal{A}$. All of these transitions are of the form $\cdot \xrightarrow{\boxed{0}} \cdot$. We check whether *end* is reachable from *init* in $\mathcal{V}$.

This is a log space reduction as there are at most 2 transitions being processed at any given time (and the space complexity required to represent a transition is logarithmic in the space representing $\mathcal{A}$). □

Claim 1. *For every $q, s \in Q$ it holds that s is reachable from q in $\mathcal{A}$ iff s is reachable from q in $\mathcal{V}$.*

Proof. Because we consider infinitely many data values, given any transition $q\xrightarrow{\boxed{a \mid g^{\in}, g^{\notin}, up, rm}} s$, where r (the register of $\mathcal{A}$) is in $g^{\notin}$, s is always reachable from q as there exists a data value d s.t. $d \notin r$. On the other hand, if $r \in g^{\in}$ then s is reachable from q iff there exists a d s.t. $d \in r$, i.e. $|r| \geq 1$. Therefore by first subtracting one when simulating such transitions in $\mathcal{V}$ we make sure that s is reachable from q in $\mathcal{V}$ iff it is reachable from q in $\mathcal{A}$, assuming the counter is equal to $|r|$ in every state $q \in Q$. □

Next, we obtain the lower bound by reducing NFA emptiness to RsA_1^{rm} emptiness.

Lemma 7. *The emptiness problem for RsA_1^{rm} is NL-hard.*

Proof. NFA emptiness, which is NL-complete, is a special case of RsA_1^{rm} emptiness (if the register is never assigned or tested, the RsA_1^{rm} can be treated as an NFA). □

Proposition 4. *The emptiness problem for RsA_1^{rm} is NL-complete.*

Proof. Follows from Lemma 6 and Lemma 7. □

5.3 Emptiness of $\mathbf{RsA}_n^{rm}$

We will parametrize the upper bound of the emptiness problem for RsA_n^{rm} on n (the number of registers). This upper bound is also applicable to RsA_n, as RsA_n^{rm} generalizes RsA_n.

Lemma 8. *The non-emptiness problem for RsA_n^{rm} is reducible to control state reachability of a 2^n-dimensional T-VASS.*

Proof. We can use the reduction used in Lemma 23 of [11], which reduces the emptiness problem for an RsA_n $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$ to coverability in a Transfer Petri Net $\mathcal{P}$ (TPN). This is done by having some places correspond to each state of $\mathcal{A}$ and others to each subset of $\mathbf{R}$. Tokens on a place corresponding to $X \subseteq \mathbf{R}$ represent the number of distinct data values stored in the registers in X and in none of the registers of $\mathbf{R}\backslash X$. Because $\mathcal{P}$ represents registers this way, it has up to 2^n transitions representing one transition of $\mathcal{A}$. The reason being that $\mathcal{P}$ takes a token from one of the places representing a subset of $\mathbf{R}$, and if the transition of $\mathcal{A}$ does not specify all registers in its guards, then $\mathcal{P}$ must have a transition for each possible placement of the input value in registers that is allowed by the guards. $\mathcal{P}$ also has two new places: *init* which non-deterministically selects one of the places representing the initial states of $\mathcal{A}$, and *fin*, which can be reached from all places representing the final states of $\mathcal{A}$. The same construction can be done for RsA_n^{rm}, with a small modification to the way transitions are constructed [11, Theorem 37].

The resulting TPN coverability problem can be viewed as a control state reachability problem of a 2^n-dimensional T-VASS $\mathcal{V}$ with the 2^n places in $\mathcal{P}$ representing subsets of $\mathbf{R}$ being the counters of $\mathcal{V}$ and the places representing the states of $\mathcal{A}$, along with the places *init* and *fin* being the control states of $\mathcal{V}$. We use a bijection $\mathit{cnt}\colon 2^{\mathbf{R}} \to \{1, \ldots, 2^n\}$ to denote the counter $\mathit{cnt}(X)$ that represents $X \subseteq \mathbf{R}$.

Transitions in a TPN are of the form $(\mathit{In}, \mathit{Out}, \mathit{Transfer})$, where *In* specifies the number of tokens taken from each place when activating a transition, *Out* specifies the number of tokens put into each place after the transition is activated, and *Transfer* is a transfer function same as on a transfer transition of a T-VASS. In transitions of $\mathcal{P}$ in particular, *In* has (except for some exceptions mentioned later) one token for a place representing a state of $\mathcal{A}$, which we will denote $\mathit{state}(\mathit{In})$, and one token for a place representing a subset of $\mathbf{R}$, which we will denote $\mathit{reg}(\mathit{In})$. The same is true for *Out*.

When simulating a transition $t^{\mathcal{P}} = (\mathit{In}_{t^{\mathcal{P}}}, \mathit{Out}_{t^{\mathcal{P}}}, \mathit{Transfer}_{t^{\mathcal{P}}})$, where $\mathit{state}(\mathit{In}) = q, \mathit{state}(\mathit{Out}) = q', \mathit{reg}(\mathit{In}) = X, \mathit{reg}(\mathit{Out}) = X'$, $\mathcal{V}$ will go through a chain of three transitions, starting in the control state q and ending in the state q' with auxiliary control states in between, in the following order:

1. *In* transition: $q \xrightarrow{\boxed{\vec{0}_d[\mathit{cnt}(X) \mapsto -1]}} \mathit{aux}_1^{t^{\mathcal{P}}}$,
2. *Transfer* transition: $\mathit{aux}_1^{t^{\mathcal{P}}} \xrightarrow{\boxed{\mathit{tr}}} \mathit{aux}_2^{t^{\mathcal{P}}}$ s.t. $\forall Y \subseteq \mathbf{R}$: $\mathit{tr}(\mathit{cnt}(Y)) = \mathit{cnt}(\mathit{Transfer}_{t^{\mathcal{P}}}(Y))$, and
3. *Out* transition: $\mathit{aux}_1^{t^{\mathcal{P}}} \xrightarrow{\boxed{\vec{0}_d[\mathit{cnt}(X') \mapsto 1]}} q'$.

$\mathcal{P}$ has some transitions that do not have one state and one register set as their *In* and *Out*. Firstly, transitions that connect *init* to places representing initial states of $\mathcal{A}$, and transitions that connect places representing final states of $\mathcal{A}$ to *fin*. None of these access registers and can thus just be transitions between states and with no effect on the counters. Secondly, the transition that ensures there are always tokens available in the place representing $\emptyset$ (i.e., values not stored in any registers). We can add the following transition to do the same in $\mathcal{V}$: $\mathit{init} \xrightarrow{\boxed{\vec{0}_d[\mathit{cnt}(\emptyset) \mapsto 1]}} \mathit{init}$. □

Proposition 5. *The emptiness problems for both* RsA_n *and* RsA_n^{rm} *are in* $\mathbf{F}_{2^n+1}$ *for* $n \geq 2$.

Proof. Given an RsA^{rm} $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$ and a $2^{|\mathbf{R}|} = d$ dimensional T-VASS $\mathcal{V}$, constructed from $\mathcal{A}$ using Lemma 8, we will ask whether the control state *fin* is reachable from the control state *init* in $\mathcal{V}$.

We do so by using the backward coverability algorithm [23], exploring bad sequences of configurations $(q_0, v_0) \dots (q_L, v_L)$, where (q_0, v_0) is a minimal final configuration and each (q_i, v_i) is a minimal configuration that can reach (q_{i-1}, v_{i-1}). In our case, $q_0 = \mathit{fin}$ and $v_0 = \vec{0}_d$. In order for this algorithm to work we need to be able to find all the possible configurations that can appear as the next element of the sequence (q_{i+1}, v_{i+1}) given the current element (q_i, v_i). We do so as follows for transitions of three different forms (special transitions either also fall in one of the categories or do not manipulate counters and are thus trivial).

1. $q_i \xrightarrow{\boxed{\vec{0}_d[k \mapsto -1]}} s$ (*In* transitions): $q_{i+1} = s$ and $v_{i+1} = v_i + \vec{0}_d[k \mapsto 1]$,
2. $q_i \xrightarrow{\boxed{\vec{0}_d[k \mapsto 1]}} s$ (*Out* transitions): $q_{i+1} = s$ and $v_{i+1} = v_i + \vec{0}_d[k \mapsto -1]$ if $v_i[k] > 0$, otherwise $v_{i+1} = v_i$, and
3. $q_i \xrightarrow{\boxed{\mathit{tr}}} s$ (*Transfer* transitions): we can take any configuration where $q_{i+1} = s$ and $\forall 1 \leq j \leq d$: $v_{i+1}[j] = \sum_{k \in \mathit{tr}(i)} v_i[k]$.

Note that we only add elements to the sequence such that the sequence stays bad. To find a bound on the length of these sequences, we need a norm and a control function. As a norm we can use the sum of all the vector's elements $|(q, v)|_{\mathcal{V}} \triangleq \sum_{i=1}^{d} v[i]$. Given that, we can use a control function g: $g(x) = x + 1$, as the norm of a configuration in the sequence can increase by at most one from the previous configuration.

As the function g is in $\mathbf{F}_1$, we can use Lemma 1 (the length function theorem) to determine that the maximum length of such a sequence is $\mathbf{F}_{2^{|\mathbf{R}|}+1}$. A non-deterministic algorithm is able to correctly guess the next element of the sequence and if it finds a configuration

$(init, \vec{0}_d)$ then the language of $\mathcal{A}$ is not empty. Thus the emptiness problem for RsA_n^{rm} is in $\mathbf{F}_{2^n+1}$. Note that for $n \geq 2$, this is already in $\mathbf{F}_5$, and thus the exponential ($\mathbf{F}_2$) construction in Lemma 8 has no bearing on the resulting complexity. □

Chapter 6

Extending Streaming Data String Transducers

In this chapter, we present a variant of SDSTs equipped with set-registers and show that its functional equivalence problem is decidable. This variant does not operate on an ordered data domain and can therefore only check for variable equality. The set-registers are used to guard transitions as usual, but can only be updated by adding in, removing in, or being left as they were.

A streaming data string transducer with set-registers (SDST$^{\text{set}}$) $\mathcal{S}^{set}$ over $\mathbb{D}$ from an input alphabet Σ to an output alphabet Γ is a tuple $(Q, q_i, V, X, \mathbf{R}, O, \Delta)$, where Q, q_i, X, O are the same as in normal SDSTs (cf. Section 2.5), V is also the same as in SDSTs, except it does not include in, $\mathbf{R}$ is a finite set of set-registers and Δ is a set of transitions of the form $q \xrightarrow{\boxed{a \mid g^{=}, g^{\neq}, g^{\in}, g^{\notin}, up}} q'$, where q, a, q' are the source state, Σ-symbol, and target state respectively, same as in SDSTs, $g^{=}, g^{\neq} \subseteq V$, where $g^{=} \cap g^{\neq} = \emptyset$ are the positive and negative variable guards respectively, $g^{\in}, g^{\notin} \subseteq \mathbf{R}$, where $g^{\in} \cap g^{\notin} = \emptyset$ are the positive and negative set-register guards respectively, and up is an update mapping of V to $V^{in} = V \cup \{in\}$, X to $((\Gamma \times V^{in}) \cup X)^*$, and $\mathbf{R}$ to $\{-1, 0, +1\}$. Furthermore, it is required that (i) for all $q \in Q, x \in X$ there is at most one occurrence of x in $O(q)$, (ii) for each $q \xrightarrow{\boxed{a \mid \varphi, up}} q' \in \Delta, x \in X$ there is at most one occurrence of x in the set of strings $\{up(y) \mid y \in X\}$, and (iii) for each $q \xrightarrow{\boxed{a \mid g_1^{=}, g_1^{\neq}, g_1^{\in}, g_1^{\notin}, up_1}} q', q \xrightarrow{\boxed{a \mid g_2^{=}, g_2^{\neq}, g_2^{\in}, g_2^{\notin}, up_2}} q'' \in \Delta$ it holds that $g_1^{=} \cap g_2^{\neq} \neq \emptyset$ or $g_2^{=} \cap g_1^{\neq} \neq \emptyset$ or $g_1^{\in} \cap g_2^{\notin} \neq \emptyset$ or $g_2^{\in} \cap g_1^{\notin} \neq \emptyset$.

A configuration of $\mathcal{S}^{set}$ is a pair (q, f), where $q \in Q$ is the current state and $f \in ((V \to (\mathbb{D} \cup \{\bot\})) \cup (X \to (\Gamma \times \mathbb{D})^*) \cup (\mathbf{R} \to 2^{\mathbb{D}}))$ is the current variable and set-register assignment. The initial configuration of $\mathcal{S}^{set}$ is the pair $c_{init} = (q_i, f_i)$, where $f_i = \{v \mapsto \bot \mid v \in V\} \cup \{x \mapsto \epsilon \mid x \in X\} \cup \{r \mapsto \emptyset \mid r \in \mathbf{R}\}$. For any variable assignment f we also define $f^{eval} \colon ((\Gamma \times V^{in}) \times X)^* \to (\Gamma \times \mathbb{D})^*$, which evaluates a right-hand side of a data string variable update. It is defined for a word $y = y_1 \dots y_n$ as $f^{eval}(y) = eval(y_1) \dots eval(y_n)$, where $eval(y_i) = (a, f(v))$ if $y_i = (a, v), a \in \Gamma, v \in V$, $eval(y_i) = f(y_i)$, if $y_i \in X$, and $eval(y_i) = (a, in)$, if $y_i = (a, in)$ for $a \in \Gamma$.

Let $c_1 = (q_1, f_1), c_2 = (q_2, f_2)$ be two configurations of $\mathcal{S}^{set}$. We say that $\mathcal{S}^{set}$ can make a step from c_1 to c_2 over (a, d) using transition $t = q_1 \xrightarrow{\boxed{a \mid g^{=}, g^{\neq}, g^{\in}, g^{\notin}, up}} q_2$, denoted as $c_1 \vdash_t^{(a,d)} c_2$ iff

1. $\forall r^{=} \in g^{=} \colon d = f_1(r^{=})$ and $\forall r^{\neq} \in g^{\neq} \colon d \neq f_1(r^{\neq})$ and $\forall r^{\in} \in g^{\in} \colon d \in f_1(r^{\in})$ and $\forall r^{\notin} \in g^{\notin} \colon d \notin f_1(r^{\notin})$,

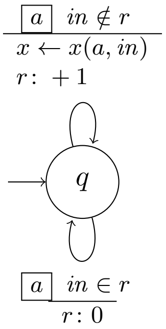


Figure 6.1: A SDST$^{\text{set}}$ performing the transduction F_{unique}, removing duplicate data values from the input data word

2. $\forall v \in V \colon f_2(v) = f'_1(up(v))$, and $\forall x \in X \colon f_2(x) = f_1'^{eval}(up(x))$, where $f'_1 = f_1[in \mapsto d]$, and

3. $\forall r \in \mathbf{R} \colon f_2(r) = \begin{cases} f_1(r) \setminus \{d\} & \text{if } up(r) = -1, \\ f_1(r) & \text{if } up(r) = 0, \text{ and} \\ f_1(r) \cup \{d\} & \text{if } up(r) = +1. \end{cases}$

The definitions for a run of $\mathcal{S}^{set}$, a transduction of $\mathcal{S}^{set}$ and the functional equivalence problem for SDST$^{\text{set}}$s are the same as for SDSTs.

Example 7. Consider the transduction F_{unique} over $\mathbb{D}$ from $\Sigma = \{a\}$ to $\Gamma = \{a\}$, that removes any data values that appeared previously in the input data word. See an SDST$^{\text{set}}$ it in Figure 6.1. It has one state q, one data string variable x, and one set-register r. Intuitively, if in is not in r, it adds it to the end of x and adds it to r. If in is already in r, it ignores it.

6.1 Deciding Functional Equivalence

The decidability of functional equivalence of SDST$^{\text{set}}$s opens up possibilities for analysis and verification of single-pass list-processing programs that use a set type data structure. We will show that the functional equivalence problem is decidable for SDST$^{\text{set}}$s by reducing it to VASS reachability. We start with the fact that VASS reachability is decidable.

Lemma 9. [5, Corollary 2] *The VASS reachability problem is Ackermann-complete.*

Next, we show that we can construct a VASS from an SDST$^{\text{set}}$ that preserves reachability. The idea is similar to the one in Lemma 8 (RsA emptiness is reducible to T-VASS control state reachability), where we track how many values are stored exactly in each subset of set-registers.

Lemma 10. *Given a SDST^{set} $\mathcal{S} = (Q^{\mathcal{S}}, q_i^{\mathcal{S}}, V, X, \mathbf{R}, O, \Delta^{\mathcal{S}})$, we can construct a $(2^{|\mathbf{R}|} - 1)$-dimensional VASS $\mathcal{V} = (Q^{\mathcal{V}}, \Delta^{\mathcal{V}}, q_i^{\mathcal{V}})$, such that for all $q \in Q^{\mathcal{S}}$ it holds that q is reachable from $q_i^{\mathcal{S}}$ in $\mathcal{S}$ iff some $(q, P, f) \in Q^{\mathcal{V}}$ is reachable from $q_i^{\mathcal{V}}$ in $\mathcal{V}$.*

Proof. First, we modify $\mathcal{S}$ so that all of its transitions are fully specified, i.e., $\forall q \xrightarrow{a \mid g^{=}, g^{\neq}, g^{\in}, g^{\notin}, up} q' \in \Delta^{\mathcal{S}}, v \in V, r \in \mathbf{R} \colon v \in g^{=} \cup g^{\neq} \land r \in g^{\in} \cup g^{\notin}$.

The states of $\mathcal{V}$ are $Q^{\mathcal{V}} \subseteq (Q^{\mathcal{S}} \times A_P \times A_f) \cup A_{aux}$, where

- $A_P = \{P \mid \bigcup_{C \in P} C \subseteq V \wedge \emptyset \notin P \wedge \forall C_1, C_2 \in P \colon C_1 \cap C_2 = \emptyset\}$,
- $A_f = \{P \rightarrow 2^{\mathbf{R}} \mid P \in A_P\}$, and
- $A_{aux} = \{aux_t^{(q,P,f)} \mid t \in \Delta^{\mathcal{S}}, q \in Q^{\mathcal{S}}, P \in A_P, f \in A_f\}$.

For all $(q, P, f) \in Q^{\mathcal{V}}$ it holds that $f \in (P \rightarrow \cdot)$. Given a $(q, P, f) \in Q^{\mathcal{V}}$, q is a state of $\mathcal{S}$, P is a partition of the defined variables of $\mathcal{S}$ based on the equality of the stored data values, and f is a function describing exactly which set-registers is the data value of each equivalence class stored in. The initial control state of $\mathcal{V}$ is $q_i^{\mathcal{V}} = (q_i^{\mathcal{S}}, \emptyset, \emptyset)$.

$\mathcal{V}$ has counters each corresponding to a non-empty subset of set-registers, we will use an arbitrary bijection $cnt \colon (2^{\mathbf{R}} \setminus \emptyset) \rightarrow \{1, \ldots, 2^{|\mathbf{R}|} - 1\}$ to relate each set to its counter. Each counter will be used to store the number of unique data values stored exactly in a set of set-registers excluding the data values also stored in variables (those are tracked in the state control of $\mathcal{V}$).

To construct $\Delta^{\mathcal{V}}$, we will take each combination of macrostate (q, P, f), and transition $q_1 \xrightarrow{\boxed{a \mid g^=, g^{\neq}, g^{\in}, g^{\notin}, up}} q_2$, where $q = q_1$ and simulate the transition from the macrostate. To simulate $t = q \xrightarrow{\boxed{a \mid g^=, g^{\neq}, g^{\in}, g^{\notin}, up}} q'$ from (q, P, f), we generate a sequence of two transitions in $\mathcal{V}$ ending in the target macrostate (q', P', f') with an auxiliary state in between. First, however, we need to compute P' and f'. We start by computing $C_{in} = \{v \mid up(v) = in \vee up(v) \in g^=, v \in V\}$, the equivalence class of variables equal to in, $R_{in} = \{r \mid up(r) = +1 \vee (r \in g^{\in} \wedge up(r) = 0)\}$, the set of registers storing the value in, and $P_{aux} = \{C \mapsto \{v \mid up(v) \in C, v \in V\} \mid C \in P \setminus \{g^=\}\}$, mapping each old equivalence class to the new equivalence class with the same data value (except for the class of variables storing in). We can then calculate $P' = \{P_{aux}(C) \mid P_{aux}(C) \neq \emptyset, C \in P\} \cup Y_{C_{in}}$, where $Y_{C_{in}} = \{C_{in}\}$ if $C_{in} \neq \emptyset$ and $Y_{C_{in}} = \emptyset$ otherwise. And $f' = \{C' \mapsto f(C) \mid (C \mapsto C') \in P_{aux}\} \cup Y_{C_{in} \mapsto R_{in}}$, where $Y_{C_{in} \mapsto R_{in}} = \{C_{in} \mapsto R_{in}\}$, if $C_{in} \neq \emptyset$ and $Y_{C_{in} \mapsto R_{in}} = \emptyset$ otherwise. The transition sequence is then as follows:

1. guard: $\begin{cases} (q, P, f) \xrightarrow{\boxed{\vec{0}}} aux_t^{(q,P,f)} & \text{if } \begin{cases} g^= \neq \emptyset \wedge g^= \in P \wedge f(g^=) = g^{\in} \text{ or} \\ g^= = g^{\in} = \emptyset, \end{cases} \\ (q, P, f) \xrightarrow{\boxed{\vec{0}[cnt(g^{\in}) \mapsto -1]}} aux_t^{(q,P,f)} & \text{if } g^= = \emptyset \neq g^{\in}, \text{ and} \\ \text{no transition} & \text{otherwise.} \end{cases}$
2. update: $aux_t^{(q,P,f)} \xrightarrow{\boxed{\vec{v}}} (q', P', f')$, where for all $R_i \in 2^{\mathbf{R}} \setminus \{\emptyset\}$ it holds that $\vec{v}[cnt(R_i)] = |\{C \mid f(C) = R_i, C \in P\}| - |\{C' \mid f'(C') = R_i, C' \in P'\}| + y$, where $y = 1$ if $C_{in} = \emptyset \wedge R_i = R_{in}$, otherwise $y = 0$.

□

Theorem 1. *The functional equivalence problem for SDST^{set} is decidable.*

Proof. Two SDST$^{\text{set}}$s $\mathcal{S}_1^{set}, \mathcal{S}_2^{set}$ are not equivalent if there exists some string w such that either only of $[\![\mathcal{S}_1^{set}]\!](w), [\![\mathcal{S}_2^{set}]\!](w)$ is defined, or $[\![\mathcal{S}_1^{set}]\!](w), [\![\mathcal{S}_2^{set}]\!](w)$ are both defined, but have different lengths, or $[\![\mathcal{S}_1^{set}]\!](w), [\![\mathcal{S}_2^{set}]\!](w)$ are defined, are of the same length, but there exists a position at which they differ.

We only show how to find a position p at which the outputs differ (the other two constructions are simpler). The construction is based on the proof of Theorem 12 in [1] (SDST functional equivalence decidability), where a 1-counter machine $\mathcal{M}$ is constructed in such a way that it simulates two SDSTs $\mathcal{S}_1, \mathcal{S}_2$ from Σ to Γ running in parallel, guesses a position at which the outputs of the two SDSTs differ during the simulation and uses the counter to check that the guess is the same for both SDSTs.

$\mathcal{M}$ stores the states of $\mathcal{S}_1, \mathcal{S}_2$ in its state control directly. For data string variables, $\mathcal{M}$ stores where it thinks each one will appear in the output in relation to the position p: (i) left of p (class L), (ii) p is in this data string (class C), (iii) right of p (class R), and (iv) does not contribute to the output (class N). $\mathcal{M}$ then non-deterministically updates its guess of which data string variable is in which class at each step, while maintaining consistency of the guesses from one step to another. When performing these guesses, $\mathcal{M}$ keeps track of the number of symbols to the left of p using the counter. One of the SDSTs adds one to the counter for each symbol left of p, while the other one subtracts, meaning that the guess of position p is the same for both SDSTs if the counter is 0.

For data variables, an order on equality classes is stored in the state control of $\mathcal{M}$, which is enough to decide reachability. Additionally, when $\mathcal{M}$ guesses that the position p of the output of $\mathcal{S}_1$ appears on a right-hand side of a data string variable assignment (this data variable would thus be added to class C), it adds a new variable vp_1 to the appropriate equivalence class of variables stored in the state control. It must also store the Γ-symbol that appears in the output at position p in its state control. The same is done for vp_2 for $\mathcal{S}_2$.

The final states of $\mathcal{M}$ are those where vp_1 and vp_2 are defined, but are in different equivalence classes or the Γ-symbols output at p differ.

Because the set-registers do not contribute to the output, we can keep most of the construction the same, except that we need a new way to determine which transitions are reachable during the simulation. We do so by constructing a VASS from the product of the two SDST$^{\text{set}}$s, as described in Lemma 10, instead of a 1-counter machine and adding the counter from the original construction to the VASS. We will call it the *position counter*.

As the position counter needs to be allowed to go below 0, which is not allowed in a VASS, we add the *supervisor counter*. The supervisor and position counters are incremented together arbitrarily many times in the initial state of the VASS. We then add a new final state f accessible (without modifying the counter values) from the states marked as final in the original construction. In f, we add a self-loop decrementing the supervisor and position counters together and self-loops decrementing each of the other counters on their own. We then check whether the configuration $(f, \vec{0})$ is reachable, thus reducing the problem to VASS reachability, which is decidable by Lemma 9.

□

SDST$^{\text{set}}$s could be generalized by allowing the values of data variables to be stored in and removed from set-registers (as opposed to just *in*). Because the construction in Lemma 10 already keeps track of which values of data variables are also stored in which set-registers, it should be easily extensible to also function for this generalization (therefore functional equivalence decidability would also hold). Because the models are deterministic, the generalization would also be more expressive than SDST$^{\text{set}}$. As an example, consider a transducer that verifies that all data values that immediately precede the symbol # (which may appear arbitrarily many times in the input) are distinct. SDST$^{\text{set}}$s are unable to check this, as they would have to guess whether or not the next symbol would be # to decide whether or not the current data value should be unique.

Chapter 7

Improvements to RA Determinisation

As RAs are non-determinisable in general (see Fact 1), DRsAs have been suggested in [11] as a means to determinise a class of NRAs. In this chapter we will first show the determinisation algorithm, and then present supplementary algorithms to enlarge the class of NRAs that it can determinise. First, let us introduce some notation and terminology.

Let $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$ be an NRA. The set $\mathbf{R}[q]$ of registers *active* in a given state $q \in Q$ is the set of registers for which there exists a transition $q_1 \xrightarrow{a \mid g^=, g^{\neq}, up} q_2 \in \Delta$, where (i) $q_2 = q$ and $up(r) \neq \bot$ or (ii) $q_1 = q$ and $r \in g^= \cup g^{\neq}$. Given a set of states S, $\mathbf{R}[S]$ denotes the set of registers active in any of the states in S. Formally, $\mathbf{R}[S] = \bigcup_{q \in S} \mathbf{R}[q]$. $\mathcal{A}$ is *register-local*, if for any two distinct states q, q', it holds that $\mathbf{R}[q] \cap \mathbf{R}[q'] = \emptyset$, i.e., no register is active in more than one state. $\mathcal{A}$ is *single-valued*, if for any reachable configuration (q, f), for any two distinct registers r, $r' \in \mathbf{R}$ it holds that $f(r) \neq f(r')$, i.e., every data value is stored in at most one register.

Any NRA can be modified to be register-local while preserving its language by creating copies of a register for every state it is active in (and modifying the updates of transitions accordingly). All NRAs can also be modified to be single-valued while preserving their language by creating copies of each state for each possible partition of $\mathbf{R}$, denoting which registers hold the same value, actually storing it in only one of them, and adjusting the transition relation to reflect this.

Algorithm 1 shows the latest version [17] of the original determinisation algorithm [11, Algorithm 1] determinising an NRA $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$. It works on a similar principle as the traditional subset construction algorithm for determinising finite automata [21], by creating macrostates of the form $(S, c) \in 2^Q \times (\mathbf{R} \to \{0, 1, \omega\})$, where S is a set of states $\mathcal{A}$ could be in at a given point and c is a mapping of registers to the number of values stored in them, with ω representing all sizes > 1. The algorithm starts from the macrostate $(I, \{r \mapsto 0 \mid r \in \mathbf{R}\})$. For each combination of a Σ-symbol a and a subset of non-empty registers g (Line 5) it then collects transitions of $\mathcal{A}$ that can be taken using (a, d), assuming d is stored in the registers in q (Line 6). The collected transitions are then processed (Lines 9 to 15), and a transition of the DRsA is generated (Line 24) to simulate them.

Lines 16 to 19 check that the generated DRsA does not overapproximate $\mathcal{A}$ by checking that each register update combination is also in the original NRA. Line 8 checks that there are no non-membership tests being performed on registers with more than one value stored in them, as it is semantically different from the non-equality test, because we are collecting

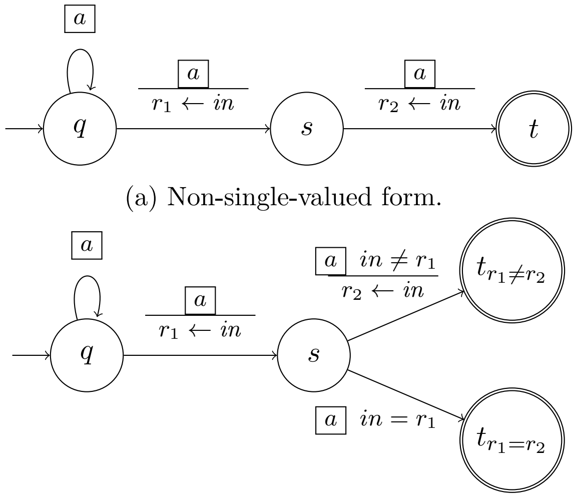
Algorithm 1: Determinisation of an NRA into a DRsA

```
Input  : Single-valued, register-local NRA $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$
Output: DRsA $\mathcal{A}' = (\mathcal{Q}', \mathbf{R}, \Delta', I', F')$ with $L(\mathcal{A}') = L(\mathcal{A})$ or $\bot$
1  $\mathcal{Q}' \leftarrow \mathsf{worklist} \leftarrow I' \leftarrow \{(I, c_0 = \{r \mapsto 0 \mid r \in \mathbf{R}\})\}$;
2  $\Delta' \leftarrow \emptyset$;
3  while $\mathsf{worklist} \neq \emptyset$ do
4      $(S, c) \leftarrow \mathsf{worklist}.pop()$;
5      foreach $a \in \Sigma, g \subseteq \{r \in \mathbf{R}[S] \mid c(r) \neq 0\}$ do
6          $T \leftarrow \{q \xrightarrow{a \mid g^=, g^{\neq}, \cdot} q' \in \Delta \mid q \in S, g^= \subseteq g, g^{\neq} \cap g = \emptyset\}$;
7          $S' \leftarrow \{q' \mid \cdot \xrightarrow{\cdot \mid \cdot, \cdot, \cdot} q' \in T\}$;
8          if $\exists q \xrightarrow{\cdot \mid \cdot, g^{\neq}, \cdot} q' \in T, \exists r \in g^{\neq} : c(r) > 1$ then return $\bot$ ;
9          $T^\bullet = \{q \xrightarrow{a \mid g^=, g^{\neq}, up[g^=/in]} q' \mid q \xrightarrow{a \mid g^=, g^{\neq}, up} q' \in T\}$;
10         foreach $r_i \in \mathbf{R}$ do
11             $tmp \leftarrow \emptyset$;
12             foreach $\cdot \xrightarrow{\cdot \mid g^=, \cdot, up} \cdot \in T^\bullet$ do
13                 if $up(r_i) = y \neq \bot \wedge c(y) \neq 0$ then $tmp \leftarrow tmp \cup \{y\}$ ;
14             $op_{r_i} \leftarrow \begin{cases} tmp \setminus \{in\} & \text{if } tmp \cap g \neq \emptyset \text{ and} \\ tmp & \text{otherwise} \end{cases}$;
15             $c'(r_i) \leftarrow \sum_{x \in op_{r_i}}^{>1 \rightsquigarrow \omega} c(x)$ ;
16         foreach $q' \in S'$ do
17             $P \leftarrow op_{r_1} \times \cdots \times op_{r_n}$ for $\{r_1, \ldots, r_n\} = \mathbf{R}[q']$;
18             foreach $(x_1, \ldots, x_n) \in P$ do
19                 if $\nexists(\cdot \xrightarrow{\cdot \mid \cdot, \cdot, up} q') \in T^\bullet$ s.t. $\bigwedge_{1 \leq i \leq n} up(r_i) = x_i$ then return $\bot$ ;
20         $up' \leftarrow \{r_i \mapsto op_{r_i} \mid r_i \in \mathbf{R}\}$;
21         if $(S', c') \notin \mathcal{Q}'$ then
22             $\mathsf{worklist}.push((S', c'))$;
23             $\mathcal{Q}' \leftarrow \mathcal{Q}' \cup \{(S', c')\}$;
24         $\Delta' \leftarrow \Delta' \cup \{(S, c) \xrightarrow{a \mid g, \mathbf{R} \setminus g, up'} (S', c')\}$;
25 return $\mathcal{A}' = (\mathcal{Q}', \mathbf{R}, \Delta', I', \{(S, c) \in \mathcal{Q}' \mid S \cap F \neq \emptyset\})$;
```

the possible values of the register. We also point out Line 9, which *collapses* the data values stored in a register to just *in*, after positively testing for membership (that simulates testing for equality in the original NRA). This collapse of registers is why single-valuedness is required. Having a register copy its value into another one could create a situation, where one of the registers is tested for membership and collapsed, while the other remains as it was (and could thus be later tested for membership of a different data value, which would be inconsistent with the original NRA)

7.1 NRA Pre-processing

The requirement of the NRA to be single-valued has been identified as limiting. Converting an NRA to an equivalent single-valued NRA can introduce non-equality guards on registers whose value is chosen non-deterministically (i.e., in two runs over the same string, the



(a) Non-single-valued form.

(b) Single-valued form.

Figure 7.1: Single valued conversion for an NRA that stores data values in two registers.

registers can have different values at the same position in the string). This will cause Algorithm 1 to return $\perp$ on Line 8.

Example 8. Consider the NRA over $\Sigma = \{a\}$ shown in Figure 7.1a. It non-deterministically selects a data value to store in r_1 and then stores the following data value in r_2. When converting it to the single-valued form, we obtain the NRA shown in Figure 7.1b, which also non-deterministically selects the data value for r_1, but before storing the next data value in r_2, it must check that the data value is not already stored in r_1. If the data value is already stored in r_1, then it marks that $r_1 = r_2$ in its state control and does not actually store anything in r_2. With this construction, however, we necessarily introduce a non-equality guard on r_1, which would cause Algorithm 1 to return $\perp$ on Line 8.

The single-valued requirement is also stricter than necessary. To avoid a situation where we would have to collapse multiple registers, we only need to ensure that there is no transition where multiple registers are assigned the same value regardless of the input. In other words, registers can hold the same value as long as it is not guaranteed that they hold the same value. To achieve that on a transition, each register must appear on a right-hand side of an update at most once, and at most one register can be updated by *in* or a register in the equality guard.

In Algorithm 2, we show an algorithm that converts any NRA so that it satisfies the conditions specified above. Intuitively, the algorithm works by tracking a *partition* of registers in each state, with all registers that certainly store the same value being in the same class. Registers are distributed into classes on Line 11, or on Line 13, if the register is not a part of an already existing class. Empty registers are not members of any class (a set of empty registers in generated separately on Line 5). In the converted automaton, the partition classes act as registers instead of the original registers (Line 17) and new updates (in the loop on Line 21) and guards (Lines 18 and 19) are generated accordingly. Because we are generating registers at the same time as transitions, we treat $r \leftarrow \perp$ updates as implicit in the constructed NRA (otherwise transitions generated early on would be missing updates for registers created later). While Algorithm 2 can potentially create more registers than were in the original NRA, the number of registers can always be reduced to at most the

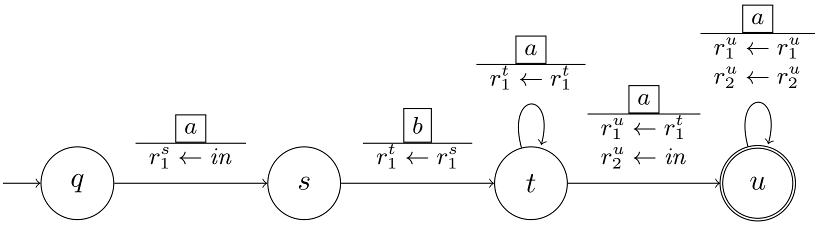


Figure 7.2: A register-local NRA that falsely triggers the overapproximation detection of Algorithm 1.

original number of registers, by using one of the registers in a class to store a value for all of them instead of having a register for each class.

7.2 DRsA Post-processing

Another limiting factor that was identified was the overapproximation detection of the original algorithm. It works by aborting the determinisation if it creates a register update combination that does not appear in the original automaton. However, in some cases where it detects an overapproximation, the automaton still accepts the same language. This happens when multiple registers in the DRsA hold the same value, and in the detected update combination that does not occur, the equal registers can be swapped around to create an update combination that does occur in the original NRA.

Example 9. Consider the NRA shown in Figure 7.2. It is in the register-local form, having multiple copies for registers r_1, r_2 (updates of the type $r \leftarrow \bot$ are implicit here). During determinisation of this NRA, when generating transitions from the macrostate $\sigma = (\{t, u\}, \{r_1^s\colon 0, r_1^t\colon 1, r_1^u\colon 1, r_2^u\colon 1\})$, registers r_1^u, r_2^u get the updates $r_1^u \leftarrow r_1^t \cup r_1^u$ and $r_2^u \leftarrow r_2^u \cup \{in\}$. This would cause Algorithm 1 to return $\bot$ on Line 19 when attepting to find one of the update combinations (i) $r_1^u \leftarrow r_1^t, r_2^u \leftarrow r_2^u$, (ii) $r_1^u \leftarrow r_1^u, r_2^u \leftarrow in$, neither of which exists in the original NRA. However, in the macrostate σ, registers r_1^t and r_1^u hold the same value, and update combination (i) is thus equivalent to $r_1^u \leftarrow r_1^u, r_2^u \leftarrow r_2^u$, and update combination (ii) is equivalent to $r_1^u \leftarrow r_1^t, r_2^u \leftarrow in$. Both of these combinations can be found in the original NRA and, therefore, the detected overapproximation was a false positive.

We propose Algorithm 3 to replace the overapproximation checking of the original automaton. The algorithm is run on the created DRsA, detects which registers hold the same values, and then checks for update combinations that are not compatible with the original NRA. It works in a similar way as the pre-processing algorithm, in that it also tracks partitions of registers for each state, with registers in the same class storing equal sets of values.

For each register, up_{aux} is created, mapping it to those classes of registers from the previous state with which it is updated (Lines 9 to 15). Then the registers that are mapped to the same set in up_{aux} are grouped together to create new classes (Lines 16 to 19). A new transition is then created with the registers renamed accordingly. Then, on Lines 28 to 33, the overapproximation test is performed using the same principle as in Algorithm 1, except the registers in the same class are interchangeable when on the right side of an update (this is reflected in the membership operator replacing the equality operator on Line 32).

Algorithm 2: Pre-processing of an NRA

```
Input  : NRA $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$
Output: Pre-processed NRA $\mathcal{A}' = (Q', \mathbf{R}', \Delta', I', F')$ with $L(\mathcal{A}') = L(\mathcal{A})$
$I' \leftarrow Q' \leftarrow$ worklist $\leftarrow \{(q, \emptyset) \mid q \in I\}$;
$\mathbf{R}' \leftarrow \emptyset$;
while worklist $\neq \emptyset$ do
    $(q, P) \leftarrow$ worklist.$pop()$;
    $C_\perp \leftarrow \{r \in \mathbf{R} \mid \nexists C \in P \colon r \in C\}$;
    foreach $q \xrightarrow{a \mid g^=, g^{\neq}, up} s \in \Delta$, where $g^= \cap C_\perp = \emptyset$ do
        $P' = \emptyset$;
        foreach $r \in \mathbf{R}$, where $up(r) = in \vee \exists C \in P \colon up(r) \in C$ do
            foreach $C' \in P'$ do
                if $\exists r' \in C' \colon \begin{cases} up(r) = up(r'),\ or \\ \exists C \in P \colon up(r), up(r') \in C,\ or \\ up(r), up(r') \in g^= \cup \{in\} \end{cases}$ then
                    $C' \leftarrow C' \cup \{r\}$;
            if $\nexists C' \colon C' \in P' \wedge r \in C'$ then
                $P' \leftarrow P' \cup \{\{r\}\}$;
        if $(s, P') \notin Q'$ then
            $Q' \leftarrow Q' \cup \{(s, P')\}$;
            worklist.$push((s, P'))$;
        $\mathbf{R}' \leftarrow \mathbf{R}' \cup \{r_{C'} \mid C' \in P'\}$;
        $g^=_{new} \leftarrow \{r_C \mid C \in P \wedge \exists r \in C \colon r \in g^=\}$;
        $g^{\neq}_{new} \leftarrow \{r_C \mid C \in P \wedge \exists r \in C \colon r \in g^{\neq}\}$;
        $up_{new} \leftarrow \emptyset$;
        foreach $C' \in P'$ do
            $tmp \leftarrow \begin{cases} in & \text{if } \exists r \in C' \colon up(r) = in, \text{ and} \\ r_C & \text{if } \exists C \in P : \forall r \in C' \colon up(r) \in C \end{cases}$;
            $up_{new} \leftarrow up_{new} \cup \{r_{C'} \mapsto tmp\}$;
        $\Delta' \leftarrow \Delta' \cup \{(q, P) \xrightarrow{a \mid g^=_{new}, g^{\neq}_{new}, up_{new}} (s, P')\}$;
$F' \leftarrow \{(q, P) \mid (q, P) \in Q' \wedge q \in F\}$;
return $\mathcal{A}'$;
```

Algorithm 3: DRsA postprocessing

```
Input  : DRsA $\mathcal{A}' = (\mathcal{Q}', \mathbf{R}, \Delta', I', F')$ potentially overapproximating the language of
         the NRA $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$
Output: DRsA $\mathcal{A}'' = (\mathcal{Q}'', \mathbf{R}'', \Delta'', I'', F'')$ with $L(\mathcal{A}'') = L(\mathcal{A})$ or $\bot$
$I'' \leftarrow \mathcal{Q}'' \leftarrow \mathsf{worklist} \leftarrow \{(\sigma', \emptyset) \mid \sigma' \in I'\}$;
$\mathbf{R}'' \leftarrow \emptyset$;
while $\mathsf{worklist} \neq \emptyset$ do
    $(\sigma, P) \leftarrow \mathsf{worklist}.pop()$;
    foreach $\sigma \xrightarrow{a \mid g^=, g^{\neq}, up} \sigma' \in \Delta'$ do
        $P' \leftarrow \emptyset$;
        $up_{aux} \leftarrow \emptyset$;
        $up_{new} \leftarrow \emptyset$;
        foreach $r \in \mathbf{R}$ do
            $Y \leftarrow \emptyset$;
            foreach $y \in up(r)$ do
                if $y \neq in$ then
                    $y \leftarrow r_C$, s.t. $C \in P \land y \in C$;
                $Y \leftarrow Y \cup \{y\}$;
            $up_{aux} \leftarrow up_{aux} \cup \{r \mapsto Y\}$;
        foreach $Y \in up_{aux}.values()$ do
            $C' \leftarrow \{r \in \mathbf{R} \mid up_{aux}(r) = Y\}$;
            $P' \leftarrow P' \cup \{C'\}$;
            $up_{new} \leftarrow up_{new} \cup \{r_{C'} \mapsto Y\}$;
        $g^=_{new} \leftarrow \{r_C \mid C \in P \land \exists r \in C \colon r \in g^=\}$;
        $g^{\neq}_{new} \leftarrow \{r_C \mid C \in P \land \exists r \in C \colon r \in g^{\neq}\}$;
        $\mathbf{R}'' \leftarrow \mathbf{R}'' \cup \{r_{C'} \mid C' \in P'\}$;
        if $(\sigma', P') \notin \mathcal{Q}''$ then
            $\mathcal{Q}'' \leftarrow \mathcal{Q}'' \cup (\sigma', P')$;
            $\mathsf{worklist}.push((\sigma', P'))$;
        $\Delta'' \leftarrow \Delta'' \cup \{(\sigma, P) \xrightarrow{a \mid g^=_{new}, g^{\neq}_{new}, up_{new}} (\sigma', P')\}$;
        $(S, c) \leftarrow \sigma'$;
        foreach $q' \in S$ do
            $U \leftarrow up_{aux}(r_1) \times \cdots \times up_{aux}(r_n)$ for $\{r_1, \ldots, r_n\} = \mathbf{R}[q']$;
            foreach $(x_1, \ldots, x_n) \in U$ do
                $(y_1, \ldots, y_n) \leftarrow (x_1, \ldots, x_n)[in/\{in\}, r_C/C]$;
                if $\nexists (\cdot \xrightarrow{\cdot \mid \cdot, \cdot, up} q') \in \Delta$ $s.t.$ $\bigwedge_{1 \leq i \leq n} up(r_i) \in y_i$ then
                    return $\bot$
$F'' \leftarrow \{(\sigma, P) \mid (\sigma, P) \in \mathcal{Q}'' \land \sigma \in F'\}$;
return $\mathcal{A}''$;
```

Chapter 8

RsA-based Regex Matching

We present a prototype RsA-based regex matcher as a way to match a class of regexes with back-references without back-tracking. The matcher works by constructing an NRA from a regex (if possible) and determinising it into a DRsA using Algorithm 1. Input words are then run on the DRsA to decide whether or not they match the regex.

8.1 Implementation

The prototype is implemented in Python. In the implementation, RAs and RsAs are a bit different from their formal definitions in Chapter 2. They do not run on data words (registers store Σ-symbols), and have sets of symbols on transitions instead of single symbols. The sets of symbols on transitions are paired with either a positive or a negative mark, with the positive mark meaning that the transition can be taken using any input symbol in the set, and the negative mark meaning that the transition can be taken using any input symbol except the ones specified in the set.

The matcher uses the regex parser of Python's RE module [18]. The result of the parser is a syntax tree, from which an NRA is created in the same way as is standard for finite automata [25] with special handling of back-references. Specifically, for the n-th (back-referenced) capture group, a transition is created with the assignment $r_n \leftarrow in$, and for a back-reference of the n-th capture group, a transition with r_n in its equality guard is created. The length of all back-referenced capture groups is checked to be one, as other capture groups are not generally representable by RAs (and the ones that are representable are typically not determinisable into DRsAs).

This automaton is then converted to its register-local form (the pre-processing in Algorithm 2 is not necessary, as no copying of values occurs in the constructed NRAs), and determinised. The determinisation is only slightly modified to accommodate the sets of values on transitions. The postprocessing algorithm (Algorithm 3) is only run if the determinisation algorithm detects overapproximation at some point. Input words are run on the resulting DRsA and if the DRsA accepts them, then the word matches the regex, otherwise it does not match the regex.

8.2 Experiments

We compared the performance of our RsA-based regex matcher against other commonly used regex matchers under simulated ReDoS attacks. For that, we extracted regexes with

Table 8.1: Numbers of RsA-compiled regexes for each ReDoS tool, and the numbers of timeouts (10 s) on the RsA-compiled regexes for each measured matcher.

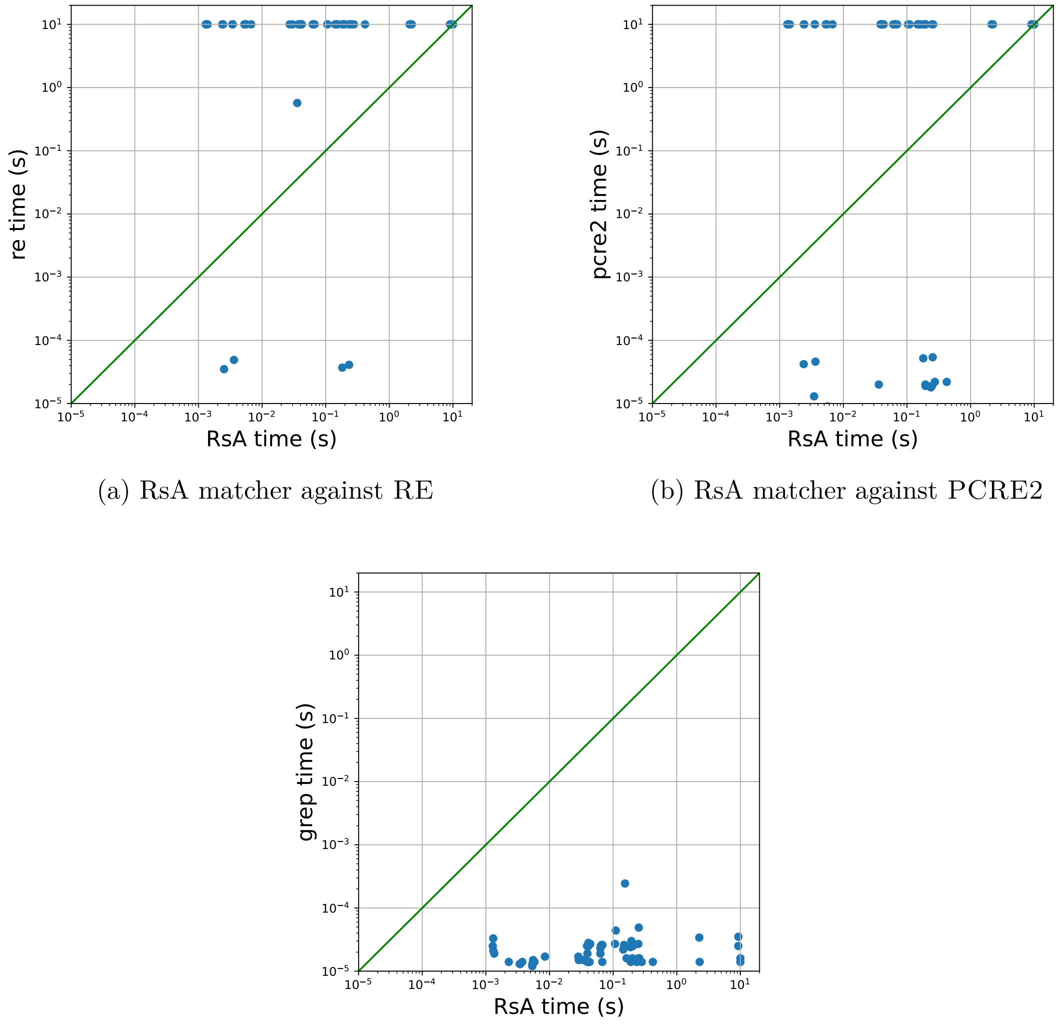
	RXXR2	Rescue
Total regexes	97	60
RsA-compiled	47	22
RsA-compilation TO	1	1
RsA TO (of RsA-compiled)	0	0
RE TO (of RsA-compiled)	43	21
PCRE2 TO (of RsA-compiled)	36	18
GREP TO (of RsA-compiled)	0	0

back-references from the production regexes collected in [6]. Out of the 537,806 unique regexes, we have found 3,091 regexes with back-references. We then used the ReDoS generator tools Rescue [24] and RXXR2 [22] to generate attack strings for the found regexes. Rescue uses a genetic algorithm combined with NFA analysis to generate attack strings (the length of attack strings is limited to 128). RXXR2 generates attack strings using static analysis of the NFA of a given regex. It generates a prefix p, a suffix s, and a pump w, which corresponds to attack strings of the form pw^*s.

Rescue has generated attack strings for 60, while RXXR2 generated attack strings for 97 of the found regexes with back-references. The RsA-based implementation's performance under the simulated ReDoS attacks was then compared with Python's RE module [18], the PCRE2 library [13], and the command line tool GNU GREP [12] (with the `-E` flag, enabling regex extensions). Note that we did not measure RE2 [8] and Hyperscan [14] as they do not support back-references. We measured the time it took each matcher to match the attack string to the corresponding regex. As RXXR2 generates attack strings of variable length depending on the number of pumps in the string, we measured each regex with 6 different strings, each with a different number of pumps. The pump numbers used were 5, 10, 20, 40, 80, and 160.

The timeout limit was set to 10 seconds. Note that PCRE2 has an internal match limit used to limit the amount of time and memory matching can use. The limit was raised such that it would not stop matching before reaching the timeout. The experiments were conducted on an Ubuntu 22.04 system with an AMD Ryzen 5 5600G @ 3.89 GHz processor and 8 GB of RAM.

Table 8.1 shows the numbers of RsA-compiled regexes (i.e., regexes for which a DRsA representing them was successfully constructed), and the numbers of timeouts each matcher had on the RsA-compiled regexes. (timeouts are counted at most once per regex for RXXR2 generated attacks). It also shows that the RsA matcher timed out during compilation of one regex for both sets of regexes (this was caused by the determinisation of the NRA representing the regex). One can see that our implementation could compile over a third of the attacked regexes for both ReDoS generators. On the RsA-compiled regexes, PCRE2 and RE time out in the majority of cases. However, GREP performs very well under attacks from both tools as it only timed out on one regex in total, across both generators, and on none of the RsA-compiled regexes.



(a) RsA matcher against RE

(b) RsA matcher against PCRE2

(c) RsA matcher against GREP

Figure 8.1: Scatter plots comparing the RsA matcher to other matchers on RsA-compiled regexes (all scatter plots show attacks by both Rescue and RXXR2)

Figure 8.1 shows scatter plots of matching times for RsA-compiled regexes of both the Rescue and RXXR2 sets. It also includes regexes where RsA compilation timed out. Out of the RXXR2 attacks, we only include the ones with 160 pumps in the attack string. Although out matcher is generally slower unless the other matcher times out, that is to be expected given that it is still a prototype and the determinisation process is not optimized.

Figure 8.2 shows the average performance of each regex matcher for each number of pumps in the attack strings generated by RXXR2 (on RsA-compiled regexes only). Again, GREP outperforms the other matchers across the board. But the graph nicely shows that RsA-based matching scales linearly with the length of input.

Both tools, Rescue and RXXR2, mostly try to target back-tracking algorithms by having the input not match the regex, but in such a way that the algorithm needs to explore a lot of possibilities before deciding that the input does not match. And even though both tools support back-references, they do not exploit them when creating attack strings.

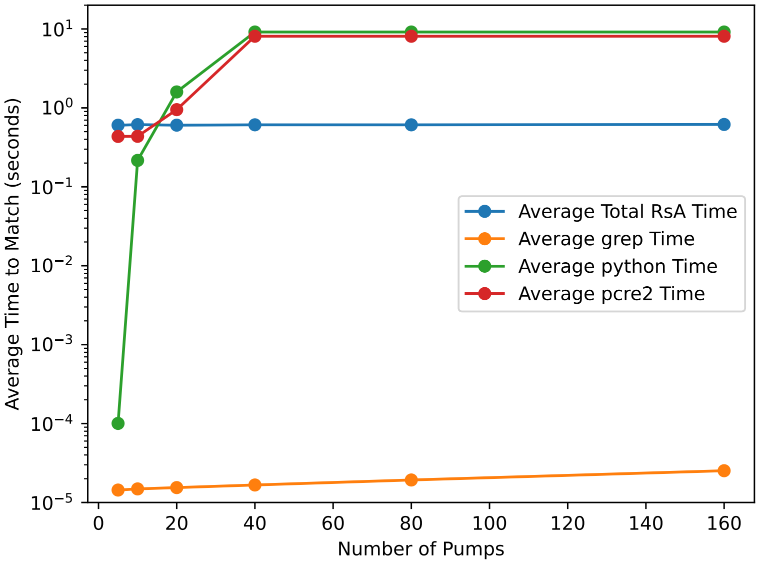


Figure 8.2: Graph showing average performance of each matcher based on the number of pumps in the RXXR2 generated attack string

We suspect the reason why GREP performs well is that it avoids back-tracking altogether and decides that the inputs do not match some other way. However, we know that it is possible to cause GREP to back-track catastrophically from a previous experiment, where we matched up the RsA matcher against GREP for regex and input that were hand-crafted to be difficult to match.

The chosen regex was `/^.*(.).*\1.*;.*;.*\1$/`, and the measured input was the string `a;a;a;a` with the first semicolon pumped up to the string `a;`^{994}`a;a;a`. This experiment was conducted on a Debian11/bullseye system, with 2 Intel Xeon X5650 @ 2.67 GHz CPUs, and 32 GB of RAM. The results are shown in Figure 8.3. One can see that on this regex and input, the RsA-based matcher drastically outperforms GREP for longer inputs. The time to match for the 1,000 character long string was just over 30 minutes for GREP, while being at about 0.16 seconds fot the RsA-based matcher.

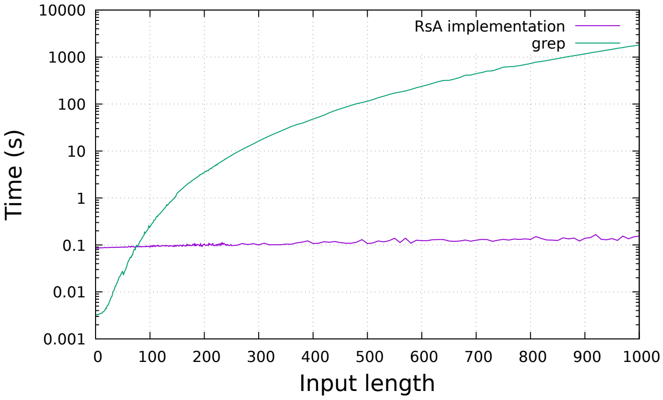


Figure 8.3: Graph of the times to match of GREP and the RsA matcher for a difficult regex and input combination

Chapter 9

Conclusion and Future Work

In this thesis, we have presented some theoretical results pertaining to extensions of register automata with set structures. Specifically, we have related the expressivity of HRAs to RsA^{rm}s, showing that every HRA can be converted to an RsA^{rm}, although it remains unclear whether the same is true in the opposite direction. We have, however, shown that DRsA^{rm}s are strictly more expressive than DHRAs.

We parametrized the emptiness problem of RsAs and RsA^{rm}s on the number of registers. Both variants have their emptiness problems NL-complete when restricted to one register. We also gave an upper bound of $\mathbf{F}_{2^n+1}$ for the emptiness problem of both variants with n registers.

Next, we have presented an extension of SDSTs, equipping them with a type of set-registers. This extension, SDST$^{\text{set}}$, is able to represent single-pass list-processing programs that use a set type data structure (such as a program that removes duplicates from a list, which cannot be represented by standard SDSTs), but it is restricted to only equality testing of data variables (as opposed to SDSTs, which can test inequality of data variables).

We have examined the existing algorithm for determinisation of NRAs into DRsAs, and found instances of NRAs that it does not determinise, even though they do have equivalent DRsAs. We have presented two algorithms to remedy the identified issues, one pre-processing the NRA before determinisation, and the other post-processing the DRsA after determinisation for a better overapproximation test.

We have also shown a practical application of RsAs in the form of a regex matcher prototype. As the regex matcher uses DRsAs to match inputs, it scales linearly in the lengths of inputs, and works for a class of regexes with back-references. It does, however, have quite a big initial overhead, as it needs to run the determinisation algorithm before matching. We have experimentally compared the prototype regex matcher to regex matchers used in practice (Python's RE module, the PCRE2 library, and GNU GREP), by subjecting them to simulated ReDoS attacks on regexes used in practice. The prototype was able to compile more than a third of the attacked regexes, and it heavily outperformed back-tracking matchers (RE and PCRE2) on these. However, the attacks were almost completely inefficient against the heavily optimized GREP, which performed better than the other matchers in all cases. We did show that even GREP can perform poorly on regexes with back-references in an experiment with a difficult hand-picked regex and input, which the RsA matcher was able to handle smoothly.

In the future, we would like to further examine extensions of SDSTs, similar to the one presented here. Specifically, we plan to examine the functional equivalence decidability of SDST$^{\text{set}}$s that keep inequality tests of variables, and to determine with what other set-register variants we could equip SDSTs while keeping their functional equivalence problem decidable. We also aim to further extend the class of regexes with back-references that can be determinised. Furthermore, we would like to start work on a ReDoS generator specifically targeting regexes with back-references.

Bibliography

[1] ALUR, R. and CERNÝ, P. Streaming transducers for algorithmic verification of single-pass list-processing programs. In: BALL, T. and SAGIV, M., ed. *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 2011, p. 599–610. DOI: 10.1145/1926385.1926454. Available at: https://doi.org/10.1145/1926385.1926454.

[2] BANERJEE, A., CHATTERJEE, K. and GUHA, S. Set Augmented Finite Automata over Infinite Alphabets. *CoRR*. 2023, abs/2311.06514. DOI: 10.48550/ARXIV.2311.06514. Available at: https://doi.org/10.48550/arXiv.2311.06514.

[3] BJÖRKLUND, H. and SCHWENTICK, T. On notions of regularity for data languages. *Theor. Comput. Sci.* 2010, vol. 411, 4-5, p. 702–715. DOI: 10.1016/J.TCS.2009.10.009. Available at: https://doi.org/10.1016/j.tcs.2009.10.009.

[4] BOJANCZYK, M., MUSCHOLL, A., SCHWENTICK, T., SEGOUFIN, L. and DAVID, C. Two-Variable Logic on Words with Data. In: *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 2006, p. 7–16. DOI: 10.1109/LICS.2006.51. Available at: https://doi.org/10.1109/LICS.2006.51.

[5] CZERWINSKI, W. and ORLIKOWSKI, L. Reachability in Vector Addition Systems is Ackermann-complete. *CoRR*. 2021, abs/2104.13866. Available at: https://arxiv.org/abs/2104.13866.

[6] DAVIS, J. C., IV, L. G. M., COGHLAN, C. A., SERVANT, F. and LEE, D. Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-use and Portability of Regular Expressions. *CoRR*. 2021, abs/2105.04397. Available at: https://arxiv.org/abs/2105.04397.

[7] DEMRI, S. and LAZIC, R. LTL with the Freeze Quantifier and Register Automata. In: *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 2006, p. 17–26. DOI: 10.1109/LICS.2006.31. Available at: https://doi.org/10.1109/LICS.2006.31.

[8] GOOGLE. *RE2* [online]. 2010 [cit. 2024-04-30]. Available at: https://github.com/google/re2.

[9] GRIGORE, R., DISTEFANO, D., PETERSEN, R. L. and TZEVELEKOS, N. Runtime Verification Based on Register Automata. In: PITERMAN, N. and SMOLKA, S. A., ed. *Tools and Algorithms for the Construction and Analysis of Systems - 19th*

International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Springer, 2013, vol. 7795, p. 260–276. Lecture Notes in Computer Science. DOI: 10.1007/978-3-642-36742-7_19. Available at: https://doi.org/10.1007/978-3-642-36742-7_19.

[10] Grigore, R. and Tzevelekos, N. History-Register Automata. *Log. Methods Comput. Sci.* 2016, vol. 12, no. 1. DOI: 10.2168/LMCS-12(1:7)2016. Available at: https://doi.org/10.2168/LMCS-12(1:7)2016.

[11] Gulčíková, S. and Lengál, O. Register Set Automata (Technical Report). arXiv. 2022. DOI: 10.48550/ARXIV.2205.12114. Available at: https://arxiv.org/abs/2205.12114.

[12] Haertel, M. et al. *GNU grep* [online]. Version 3.6. September 2022 [cit. 2024-04-30]. Available at: https://www.gnu.org/software/grep/.

[13] Hazel, P. *Perl-compatible Regular Expressions* [online]. Version 10.42. December 2022 [cit. 2024-04-30]. Available at: https://www.pcre.org/.

[14] Intel. *Hyperscan* [online]. 2015 [cit. 2024-04-30]. Available at: https://github.com/intel/hyperscan.

[15] Jones, N. D. Space-Bounded Reducibility among Combinatorial Problems. *J. Comput. Syst. Sci.* 1975, vol. 11, no. 1, p. 68–85. DOI: 10.1016/S0022-0000(75)80050-X. Available at: https://doi.org/10.1016/S0022-0000(75)80050-X.

[16] Kaminski, M. and Francez, N. Finite-Memory Automata. *Theor. Comput. Sci.* 1994, vol. 134, no. 2, p. 329–363. DOI: 10.1016/0304-3975(94)90242-9. Available at: https://doi.org/10.1016/0304-3975(94)90242-9.

[17] Lengál, O. *Personal communication.* January 2023.

[18] Lundh, F. and Kuchling, A. M. *Python Standard Library: re module* [online]. Version 3.10.12. June 2023 [cit. 2024-04-30]. Available at: https://docs.python.org/3/library/re.html.

[19] Minsky, M. L. Recursive Unsolvability of Post's Problem of „Tag“ and other Topics in Theory of Turing Machines. *Annals of Mathematics.* Annals of Mathematics. 1961, vol. 74, no. 3, p. 437–455. ISSN 0003486X. Available at: http://www.jstor.org/stable/1970290.

[20] Neven, F., Schwentick, T. and Vianu, V. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.* 2004, vol. 5, no. 3, p. 403–435. DOI: 10.1145/1013560.1013562. Available at: https://doi.org/10.1145/1013560.1013562.

[21] Rabin, M. O. and Scott, D. S. Finite Automata and Their Decision Problems. *IBM J. Res. Dev.* 1959, vol. 3, no. 2, p. 114–125. DOI: 10.1147/RD.32.0114. Available at: https://doi.org/10.1147/rd.32.0114.

[22] Rathnayake, A. and Thielecke, H. Static Analysis for Regular Expression Exponential Runtime via Substructural Logics. *CoRR*. 2014, abs/1405.7058. Available at: http://arxiv.org/abs/1405.7058.

[23] Schmitz, S. and Schnoebelen, P. *Algorithmic Aspects of WQO Theory*. Master. France, 2012. Lecture. Available at: https://cel.hal.science/cel-00727025.

[24] Shen, Y., Jiang, Y., Xu, C., Yu, P., Ma, X. et al. ReScue: crafting regular expression DoS attacks. In: Huchard, M., Kästner, C. and Fraser, G., ed. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, 2018, p. 225–235. DOI: 10.1145/3238147.3238159. Available at: https://doi.org/10.1145/3238147.3238159.

[25] Sipser, M. *Introduction to the Theory of Computation*. 2nd ed. Thomson Course Technology, 2006. ISBN 0-534-95097-3.

[26] Touili, T. Register Automata for Malware Specification. In: *ARES 2022: The 17th International Conference on Availability, Reliability and Security, Vienna,Austria, August 23 - 26, 2022*. ACM, 2022, p. 147:1–147:7. DOI: 10.1145/3538969.3544442. Available at: https://doi.org/10.1145/3538969.3544442.

[27] Tzevelekos, N. Fresh-register automata. In: Ball, T. and Sagiv, M., ed. *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 2011, p. 295–306. DOI: 10.1145/1926385.1926420. Available at: https://doi.org/10.1145/1926385.1926420.

[28] Weidman, A. et al. Regular expression Denial of Service - ReDoS. *OWASP Foundation* [online]. 2024 [cit. 2024-04-30]. Available at: https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS.

Appendix A

Contents of the Included Storage Media

```
/
├── src/ ............................................. Implementation source files
│   ├── rsaregex/ ........... Python module implementing RsA-based regex matching
│   ├── README.md ...................................................... Usage guide
│   └── rsa-matcher.py ....... Implementation of a grep-like program using rsaregex
├── tex-src/ .................................... LaTeX source files of this thesis
└── thesis.pdf ......................................... This thesis in pdf version
```