Relations and Transductions Realized by Restarting Automata

Dissertation zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
im Fachbereich Elektrotechnik/Informatik der Universität Kassel

vorgelegt von
Norbert Hundeshagen

Kassel im Januar 2013

Erster Gutachter:	Prof. Dr. Friedrich Otto
Zweiter Gutachter:	Prof. Dr. Martin Kutrib
Tag der Disputation:	17. Mai 2013

Zusammenfassung

Gegenstand der vorliegenden Arbeit ist die Analyse verschiedener Formalismen zur Berechnung binärer Wortrelationen. Dabei ist die Grundlage aller hier ausgeführten Betrachtungen das Modell der Restart-Automaten, welches 1995 von Jancar et. al. eingeführt wurde. Zum einen wird das bereits für Restart-Automaten bekannte Konzept der *input/output-* und *proper-Relationen* weiterführend untersucht, sowie auf *Systeme von zwei parallel arbeitenden und miteinander kommunizierenden Restart-Automaten* (PC-Systeme) erweitert. Zum anderen wird eine Variante der Restart-Automaten eingeführt, die sich an klassischen Automatenmodellen zur Berechnung von Relationen orientiert. Mit Hilfe dieser Mechanismen kann gezeigt werden, dass einige Klassen, die durch input/output- und proper-Relationen von Restart Automaten definiert werden, mit den traditionellen Relationsklassen der *Rationalen Relationen* und der *Pushdown-Relationen* übereinstimmen. Weiterhin stellt sich heraus, dass das Konzept der parallel kommunizierenden Automaten äußerst mächtig ist, da bereits die Klasse der proper-Relationen von monotonen PC-Systemen alle *berechenbaren Relationen* umfasst. Der Haupteil der Arbeit beschäftigt sich mit den so genannten Restart-Transducern, welche um eine Ausgabefunktion erweiterte Restart-Automaten sind. Es zeigt sich, dass sich insbesondere dieses Modell mit seinen verschiedenen Erweiterungen und Einschränkungen dazu eignet, eine umfassende Hierarchie von Relationsklassen zu etablieren. In erster Linie seien hier die verschiedenen Typen von monotonen Restart-Transducern erwähnt, mit deren Hilfe viele interessante neue und bekannte Relationsklassen innerhalb der *längenbeschränkten Pushdown-Relationen* charakterisiert werden. Abschließend wird, im Kontrast zu den vorhergehenden Modellen, das nicht auf Restart-Automaten basierende Konzept des *Übersetzens durch Beobachtung* („Transducing by Observing“) zur Relationsberechnung eingeführt. Dieser, den Restart-Transducern nicht unähnliche Mechanismus, wird im weitesten Sinne dazu genutzt, einen anderen Blickwinkel auf die von Restart-Transducern definierten Relationen einzunehmen, sowie eine obere Schranke für die Berechnungskraft der Restart-Transducer zu gewinnen.

Abstract

In the present thesis we introduce several ways of extending restarting automata to devices for realizing binary (word) relations, which is mainly motivated by linguistics. In particular, we adapt the notion of *input/output-relations* and *proper-relations* to restarting automata and to *parallel communicating systems of two restarting automata* (PC-systems). Further, we introduce a new model, called *restarting transducer*. Concerning input/output- and proper-relations we show equivalences of relation classes defined by restarting automata to the well-known classes of *rational relations* and *pushdown relations*. Further, regarding the proper-relations realized by certain PC-systems of two restarting automata, we show that this class includes *all computable relations*. The main part of this work concerns restarting transducers. A restarting transducer is a restarting automaton equipped with an output function. In this way we show that the hierarchy of language classes defined by restarting automata can be partly transfered to a corresponding hierarchy of relation classes. Moreover, by using results from the concepts introduced before, we prove that monotone types of restarting transducers define a *hierachy of relation classes within the length-bounded pushdown relations*. We conclude this thesis by establishing an upper bound and a different point of view on relations realized by restarting transducers through the concept of *Transducing by Observing*.

Publications

Parts of this thesis have already been published in the following refereed proceedings (chronologically ordered):

[HOV10] Norbert Hundeshagen, Friedrich Otto, and Marcel Vollweiler. Transductions Computed by PC-Systems of Monotone Deterministic Restarting Automata. In Michael Domaratzki and Kai Salomaa, editors, *CIAA*, volume 6482 of *Lecture Notes in Computer Science*, pages 163–172. Springer, 2010

[HL10] Norbert Hundeshagen and Peter Leupold. Transducing by Observing. In Henning Bordihn, Rudolf Freund, Markus Holzer, Thomas Hinze, Martin Kutrib, and Friedrich Otto, editors, *NCMA*, volume 263 of *books@ocg.at*, pages 85–98. Österreichische Computer Gesellschaft, 2010

[HO11] Norbert Hundeshagen and Friedrich Otto. Characterizing the Regular Languages by Nonforgetting Restarting Automata. In Giancarlo Mauri and Alberto Leporati, editors, *Developments in Language Theory*, volume 6795 of *Lecture Notes in Computer Science*, pages 288–299. Springer, 2011

[HO12a] Norbert Hundeshagen and Friedrich Otto. Characterizing the Rational Functions by Restarting Transducers. In Adrian Horia Dediu and Carlos Martín-Vide, editors, *LATA*, volume 7183 of *Lecture Notes in Computer Science*, pages 325–336. Springer, 2012

[HL12] Norbert Hundeshagen and Peter Leupold. Transducing by Observing and Restarting Transducers. In Rudolf Freund, Markus Holzer, Bianca Truthe, and Ulrich Ultes-Nitsche, editors, *NCMA*, volume 290 of *books@ocg.at*, pages 93–106. Österreichische Computer Gesellschaft, 2012

In addition, the following submitted works should be mentioned:

[HO12b] Norbert Hundeshagen and Friedrich Otto. Restarting Transducers, Regular Languages, and Rational Relations. Submitted for publication, June 2012

[HL13] Norbert Hundeshagen and Peter Leupold. Transducing by Observing Length-Reducing and Painter Rules. Submitted for publication, January 2013

In particular, the following list gives an overview on the already published or submitted results that are presented in this thesis. Additionally, the main contributions of the various co-authors to these results are named.

[HO11] contains the proceeding versions of the results shown in Subsection 2.2.1, where the main techniques used to prove Theorem 2.2.11, Theorem 2.2.12 and Theorem 2.2.14 have been presented and are due to Friedrich Otto.

[HOV10] contains the proceeding versions of the results shown in Section 3.2 on PC-Systems of restarting automata. There, a slightly rewritten version of Theorem 3.2.4, a part of the equivalences derived in Corollary 3.1.14, and the technique of how to prove that R_{pal} is not computable by the PC-System defined in this section (cf. Proposition 3.2.2) are due to Friedrich Otto.

[HO12a] contains a proceeding version of the main part of Section 4.3, excluding the results obtained in the Summary.

[HO12b] contains the full proofs presented in Subsection 2.2.1 and Section 4.3 (again without the Subsection 4.3.3). There, additionally to the first mark in this list, some technical formulations in Proposition 2.2.15 as well as in the Propositions 4.2.7, 4.3.4, and 4.3.5 are due to Friedrich Otto.

[HL10] contains Theorem 5.3.3. However, the proof presented in this thesis is new.

[HL12] contains the results presented in Section 5.2, where most of the proofs are adjusted or rewritten to fit the present scope. Additionally, the decreasing systems from [HL12] are here called "length-reducing systems", which refers to the more common name for the used kind of rules (see [BO93]). Furthermore, the idea of introducing a morphism instead of showing an equivalence of length-reducing systems and restarting transducers, directly (cf. Proposition 5.2.5), as well as some basic suggestions on the principle of Transducing by Observing are due to Peter Leupold.

Finally, [HL13] contains the results derived in Chapter 5.

Acknowledgements

First of all, I wish to thank my supervisor Prof. Dr. Friedrich Otto for his continuous support and guidance while working on the present topic and for providing me with the valuable grade of freedom I needed. Moreover, he substantially improved the content of this thesis, both, by carefully reading and correcting the manuscript as well as by participating as a co-author in several publications.

Secondly, I would like to thank Prof. Dr. Martin Kutrib for refereeing this thesis and for his helpful comments on the present content.

I also wish to thank my current and former colleagues for many fruitful discussions and various work-related hints. In particular, Marcel Vollweiler for sharing the process and the experiences made during writing a thesis, which hopefully helped him as much as me. Especially, Dr. Peter Leupold who contributed to my time spent on this work, scientifically and, equally important, non-scientifically. Furthermore, he suggested to compare the principle of Computing by Observing and restarting transducers. And not least, Prof. Dr. Martin Lange for giving me the opportunity to finish this thesis in the university environment.

In this context also the anonymous referees of the publications cited before should be mentioned, whose comments helped to improve the content of this work.

In some sense many people should be named here. Among those, I am indebted to my friends and family. Especially, I wish to thank my parents and my sister for supporting me in any decision I made. And most important, I would like to thank Petra for not getting lost on the journey.

Contents

List of Figures

Chapter 1

Introduction

In the 1950's Noam Chomsky introduced several types of generative grammars as a formal framework for producing sentences of natural languages. It is well known that these types of grammars describe a hierarchy of certain families of plain sets of words (i.e. formal languages), the Chomsky Hierarchy. Since then, a multitude of mechanisms for defining formal languages was invented in the field of Formal Language Theory. Most of them follow the principle of either generating (e.g. grammars) or accepting (e.g. automata) a formal language.

At about the same time that Chomsky emerged his work, several authors considered models to define sets of tuples of words (i.e. word relations). In this context, the remarkable works of Mealy [Mea55], Moore [Moo56] and Rabin and Scott [RS59] can be seen as a starting point of studies on mechanisms for (binary) word relations in the sense of Chomsky's hierarchy of languages. However, in contrast to Chomsky's contribution, the references above were not based on linguistic motivations. The latter fact changed as it turned out that formal descriptions of phenomena of natural languages mostly require more complicated structures than plain sets of words. Hence, the systematic extension of well-known models from the Chomsky Hierarchy to mechanisms for realizing binary word relations gains much interest in theoretical as well as practical contexts, which will be outlined in the following.

Starting from a theoretical point of view, there are several ways of associating relations to language generating or accepting devices, according to the main principles for computing formal languages. Mainly one might distinguish between transducers, which are automata that realize transductions, that is, they accept an input word and produce an output word, two tape automata, which process the input word on the first tape and the output word on

the second tape, and grammar based approaches, as for instance grammars that generate the output and input words in parallel. Hence, based on the capabilities of the underlying device, which normally is a model for accepting or generating a formal language, the derived mechanisms for computing relations yield a hierarchy of classes of relations.

From a practical point of view, several of the previously mentioned mechanisms for word relations proved to be reasonable for certain applications in natural language processing, speech recognition and also in formal language applications, such as compiler architecture. Just to mention only a few examples of the fields named above, some types of transducers are suitable tools for morphological analyzation of natural languages [Moh97] as well as for mirroring phonological rules [KK94].

Accordingly, the present thesis contributes to the field described above by investigating several ways of associating binary word relations to a quite recent model for accepting formal languages, the restarting automaton. In particular, the work covers three different extensions of restarting automata. Basically, these new types of machines can be classified according to the principles of how a relation can be associated to a device working on plain sets of words, which has been introduced above. Therefore, the scope of these kinds of automata covers restarting transducers, restarting automata accepting relations and a type of two-tape restarting automata (i.e. parallel communicating systems of two restarting automata). All of these extensions seem to be promising, both, from a theoretical as well as a practical point of view, for the following reasons.

In recent years restarting automata developed as a vivid research topic in Formal Language Theory. Many restrictions and extensions of these machines were considered and investigated (e.g. [Ott06]). This led to characterizations by restarting automata for nearly every important language class within the Chomsky Hierarchy. Consequently, a systematic study of restarting transducers based on the various types of restarting automata might serve as a framework for a hierarchy of relation classes according to the Chomsky Hierarchy. Such a taxonomy seems quite reasonable, as there are only a few comprehensive works on transducer extensions of well-known types of automata (e.g. [GR66, Ber79, CI83]).

Secondly, the connection of restarting automata to binary word relations is motivated by the main linguistic application of these types of machines, the Analysis by Reduction. Simply, this technique is a method to verify the (syntactical) correctness of a given sentence of a natural language. It is recommended for "free word order" languages, such as many Slavic languages (e.g. Czech, Sorbian, Russian, etc.). Accordingly, a given sentence is stepwise simplified under the condition that every step preserves the correctness or incorrectness

of the sentence processed. Therefore, restarting automata, as emerged in 1995 [JMPV95], intend to mirror this process by applying a stepwise simplification of a given input. The computation of such a machine simply terminates by accepting or rejecting after a certain number of simplification steps. From a linguistic point of view, the verification of correctness (or incorrectness) is not the only goal of performing Analysis by Reduction, as it is also a useful tool to gain deeper information on the structure of sentences of natural languages, such as word dependency information [LPK05] and morphological ambiguities [PLO03]. Hence, a general study of mechanisms that mirror the analyzation as well as the information extracting part of Analysis by Reduction seems valuable. Consequently, investigating the possibilities of how binary word relations can be associated to restarting automata might serve as a starting point for new linguistical insights.

Outline

The main goal of this thesis is to define and investigate several ways of associating binary word relations to restarting automata. For that we describe three scenarios, which are mainly motivated by linguistics and/or traditional models for computing relations.

In Chapter 2, to begin with, we recall fundamental concepts necessary for reading and understanding the thesis. In particular, we define the notions of binary word relations, transducers and restarting automata and provide their most valuable properties. Additionally, a first novel contribution is established, that is, we show three characterizations of the class of regular languages by monotone and non-forgetting restarting automata. These results somehow serve as a starting point for reflections on restarting automata and rational relations in Chapter 4.

Motivated by the disambiguation process of sentences in the Czech language [LPS07], we introduce the notions of input-output relations and proper relations in Chapter 3. This first concept of associating a binary word relation to a restarting automaton was suggested by Otto in [Ott10]. Here a relation is realized by projecting each word that is accepted by a restarting automaton to a distinguished input and output alphabet. According to the notion of relation characterizing languages [AU69], we show that depending on the restarting automata used, the class of proper and input-output relations coincides with some well-known relation classes (i.e. rational relations, pushdown relations).

In the second part of Chapter 3, we introduce a different perspective on relations of the above type. Thus, we associate input-output and proper relations to parallel communi-

cating systems of two restarting automata, which can be motivated as a kind of two tape automata characterization of relations. This leads to the result that the concept of proper relation associated to communicating automata is much too powerful, as we already are able to characterize every computable relation by the proper relation of two monotone machines.

Chapter 4 serves as the main part of the present thesis. There we introduce the so-called restarting transducer as an extension of restarting automata. Based on the basic definitions of this model we then establish some general properties, that is, error and correctness preserving for restarting transducers and the length-boundedness of relations computed by these machines. Further, we try to establish a hierarchy of restarting transducer relations according to the well-known hierarchy of language classes computed by restarting automata. Due to the fact that not all results easily carry over from automata to transducers, this only partially works for non-restricted types of restarting automata. However, the restriction to monotone restarting transducers and restarting transducers with window size one leads to some quite interesting connections between these machines and traditional relation classes. In particular, we define a restarting transducer hierarchy within the pushdown relations, where it turns out that the restriction of the window size leads to a restarting transducer characterization for several subclasses of the rational relations. Finally, we discuss the property of closure under composition, which is of interest for transducers in general, and two decision problems for the previously introduced restarting transducers.

To find new approaches for a question left open in Chapter 4, we briefly introduce a different perspective on transductions in Chapter 5 by so-called "transducing observer systems". The notion of Transducing by Observing is originally based on the method of Computing by Observing, which was introduced to mirror the way in which information is gained in biological or chemical experiments [CL03]. By extending this mechanism we derive a quite unconventional model for realizing transductions. However, as both, restarting automata and transducing observer systems can be interpreted as string rewriting systems controlled by regular languages [NO00], it turns out that in some cases the latter can be seen as a weakened version of restarting transducer. To say it in advance, the motivating open question remains unsolved. Nevertheless, Chapter 5 offers new aspects on relations computed by restarting transducers.

Related Work

Concerning the topic of this thesis the following related works are known.

As already mentioned in the outline above, the notion of input/output- and proper-relations of restarting automata (see Chapter 3) was considered in [Ott10].

Parallel to the notion of restarting transducers (see Chapter 4), the concept of restarting automata with output can also be found in [PML10b, PML10a, LMP10]. Motivated by a language descriptive system for the Czech language (the Functional Generative Description), the authors introduced a special type of restarting automaton that is enhanced to produce tree structures, which mirror dependency trees of sentences of natural languages. However, restarting automata that simply produce strings, as introduced in Chapter 4, have not been introduced yet.

The idea of observing string rewriting systems was introduced in [CL06] for accepting formal languages. To the best of the author's knowledge, using these systems to realize transductions (see Chapter 5) has not been considered yet.

Chapter 2

Preliminaries

In this chapter we lay the framework that is needed for reading and understanding the present work. It is divided into three sections, where the first one offers a standardized notation. Sections two and three concern the main definitions and results on restarting automata and on the classical theory of transducers. Although all necessary terms are explained, the present chapter is not meant to be a complete survey of the topics mentioned above. For that and for further reading we will refer to the appropriate literature within the text.

2.1 Words, Languages and Relations

Although we assume that the reader is familiar with basic terminology in mathematics and formal language theory, we recall some fundamental concepts. For further reading we recommend standard textbooks, such as [HU79], [Har78] and [RS97].

Basic Notations

In the following $\mathbb{N}$ denotes the set of natural numbers, where we assume that $0 \in \mathbb{N}$. $\mathfrak{P}(M)$ describes the power set of a set M and $\mathfrak{P}_{fin}(M)$ are all finite subsets of M. We call a set of letters/symbols an *alphabet*, usually denoted by capital Greek letters such as Σ. A *word* is defined as a finite concatenation $(\cdot)$ of symbols from Σ. By ε we denote the empty word. All non-empty words (Σ^+) and ε form the set of all words, denoted as Σ^*. From an algebraic point of view, Σ^* and the binary operation $\cdot$ define a *finitely generated*

free monoid. In this context, a monoid is a semi-group (a set paired with an associative operation) with a neutral element, ε in our case. Therewith Σ is the finite set of generators, and since all elements in Σ^* are generated in a unique manner, we call this monoid free. Finally, let $w \in \Sigma^*$ be a word, then $|w|$ denotes the length of the word, $|w|_a$ denotes the number of occurrences of the letter a in w, and by w^R we denote the reversal of a word w.

A *(formal) language* L is defined as a subset of Σ^*. As languages are sets of words, we should name some standard operations on languages, such as *union*, *intersection,* and *complement*. Further, the *concatenation* and *reversal* naturally extend from words to languages, and the closure under concatenation (Kleene Star or star-operation) is denoted by *. A not so common operation on languages is the *shuffle*. Let $u \in \Sigma^*$ and $v \in \Delta^*$ be words, then $\mathsf{sh}(u, v)$ denotes the set of all words of the form $u_1v_1u_2v_2 \cdots u_nv_n$, where $n \geq 1$, $u_1, u_2, \ldots, u_n \in \Sigma^*$, $v_1, v_2, \ldots, v_n \in \Delta^*$, such that $u = u_1u_2 \cdots u_n$ and $v = v_1v_2 \cdots v_n$. This operation extends to languages in the usual way. Thus, $\mathsf{sh}(L_1, L_2) = \bigcup_{u \in L_1,\, v \in L_2} \mathsf{sh}(u, v)$. Finally, a *morphism* is a mapping between two sets of words $\varphi : \Sigma^* \to \Delta^*$, where $\varphi(uv) = \varphi(u) \cdot \varphi(v)$ for all $u, v \in \Sigma^*$. Note that the latter property implies that also $\varphi(\varepsilon) = \varepsilon$ holds. Again the operation of applying a morphism extends naturally to a language $L \in \Sigma^*$, that is, $\varphi(L) = \{\varphi(u) \mid u \in L\}$.

Relations over Words

Since relations instead of plain languages play the major role in this work, we already introduce some basic facts at this point. Simply a (binary) relation over words is defined as a subset of the Cartesian product of two sets of words. Formally, let Σ and Δ be two alphabets, then $R \subseteq \Sigma^* \times \Delta^*$ is called a *binary (word) relation*. Obviously, R is a set of pairs of words. In the following these objects are often simply called relations, since it is clear from the context, when we are talking about word relations.

Several operations on languages easily extends to relations. Accordingly the definition for *union*, *intersection,* and *complement* is clear. The notion of *concatenation* is extended to pairs of words such that the concatenation of (u_1, v_1) and (u_2, v_2) is defined as: $(u_1, v_1) \cdot (u_2, v_2) = (u_1u_2, v_1v_2)$. This again easily extends to sets of pairs. Additionally, we should mention another operation here, unique for relations. Let $R_1 \subseteq \Sigma^* \times \Delta^*$ and $R_2 \subseteq \Delta^* \times \Gamma^*$, then the *composition* of R_1 and R_2 is defined as: $R_2 \circ R_1 = \{(u, v) \in \Sigma^* \times \Gamma^* \mid \exists x \in \Delta^* : (u, x) \in R_1 \text{ and } (x, v) \in R_2\}$.

Next we define some properties of relations.

Definition 2.1.1. *A relation R is called length-preserving if for each pair $(u, v) \in R$, $|u| = |v|$ holds.*

Definition 2.1.2. *A relation R is called length-bounded if there is an integer $c \in \mathbb{N}$, such that for each pair $(u, v) \in R$ with $u \neq \varepsilon$, $|v| \leq c \cdot |u|$.*

Finally, we introduce the notion of single valued relations. For that the sets $dom(R) = \{u \mid (u, v) \in R\}$ and $ra(R) = \{v \mid (u, v) \in R\}$ are called *domain* and *range* of a relation R.

Definition 2.1.3. *A relation R is called single valued if for all u in the domain of R, there is a unique v, such that $(u, v) \in R$.*

Hence, single valued relations are actually (partial) functions. Further information on binary word relations is provided in Section 2.3.

Grammars and Automata

The main topic in formal language theory concerns the investigation of finite representations for certain sets of words. Among the various ways of representing languages, we introduce the most common concepts here. For that we may distinguish between generating and accepting devices.

Grammars, which were defined by Chomsky in the 1950s, are the best known representative for the concept of generating sets of words. Hence, a *(generalized phrase-structure) grammar* is a four tuple $G = (V, \Sigma, P, S)$, where V is a finite alphabet of *non-terminals*, Σ is a finite alphabet of *terminals*, $V \cap \Sigma = \emptyset$, $S \in V$ is the *start-symbol* and $P \subseteq (V \cup \Sigma)^* \times (V \cup \Sigma)^*$ is the finite set of *production rules*, where for each pair $(l, r) \in P$ it holds that $|l|_V \geq 1$. Note that we write the ordered pairs $(l, r) \in P$ as *rewrite rules* $l \to r$.

With a grammar $G = (V, \Sigma, P, S)$ we associate a binary relation $\Rightarrow_G$, called the *derivation relation*. Let $u, v \in (V \cup \Sigma)^*$, then $u \Rightarrow_G v$ if and only if there are factorizations $u = xly$ and $v = xry$, such that $(l \to r) \in P$. Thus, the application of one rule transforms u into v. The reflexive and transitive closure of $\Rightarrow_G$ is denoted by $\Rightarrow_G^*$, and finally the *language generated* by G is defined as $L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$.

A grammar $G = (V, \Sigma, P, S)$ is called

- (right-)regular, if all rules in P are of the form $l \to r$, with $l \in V$ and $r \in \Sigma^* \cup \Sigma^* \cdot V$;

- context-free, if all rules in P are of the form $l \to r$, with $l \in V$ and $r \in (V \cup \Sigma)^*$;
- context-sensitive, if all rules in P are of the form $u_1 l u_2 \to u_1 r u_2$, with $l \in V$, $u_1, u_2 \in (V \cup \Sigma)^*$ and $r \in (V \cup \Sigma)^+$; additionally the production $S \to \varepsilon$ may be contained in P, then S does not appear in any right-hand side of any rule.
- monotone, if all rules in P are of the form $l \to r$, with $l, r \in (V \cup \Sigma)^*$ and $|l| \leq |r|$; additionally the production $S \to \varepsilon$ may be contained in P, then S does not appear in any right-hand side of any rule.

With each type of grammar previously introduced, we associate a *language class*, that is, the set of all languages that can be generated by a grammar of a particular type. The class of languages defined in that way are named according to their grammars. Hence, we have the class of regular (REG for short), context-free (CFL for short), context-sensitive languages (CSL for short) and the class defined by a general grammar, which is the class of recursively enumerable languages (RE for short). Further note that the concepts of being monotone and context sensitive introduced above, are equivalent from a language theoretic point of view.

Finally we may introduce two normal forms for context-free grammars. First of all, a context-free grammar $G = (V, \Sigma, P, S)$ is in *Chomsky normal form*, if for all rules $l \to r$ in P, $l \in V$ and $r \in \Sigma^* \cup V^2$ holds. Secondly a context-free grammar $G = (V, \Sigma, P, S)$ is in *Greibach normal form*, if for all rules $l \to r$ in P, $l \in V$ and $r \in \Sigma \cdot V^*$holds. Furthermore, G is in *quadratic Greibach normal form* if G is in Greibach normal form and the number of non-terminals on the right side of every rule is at most two. Additionally, sometimes the rule $S \to \varepsilon$ is required in the above normal forms to derive the empty word.

Next we turn to automata. Instead of generating a word, automata accept words, that is, they can be seen as answering the question of whether a given word belongs to a language, which is known as the *word problem*. Hence, automata accept or (possibly) reject words. This is obviously another way to characterize languages. In fact, for every language class introduced above, there is a type of automata that characterizes this set of languages.

For REG the corresponding device is a finite state automaton. A *deterministic finite state automaton* $A = (Q, \Sigma, \delta, q_0, F)$ (DFA for short) is a 5-tuple, where Q is a set of states, Σ is a finite input alphabet, q_0 is the initial state, F is the set of final states, and the transition function δ is defined as

$$\delta : Q \times \Sigma \to Q.$$

By δ^* we denote the natural extension of δ to words over Σ^*, that is, $\delta^*(q,\varepsilon) = q$ and $\delta^*(q,ua) = \delta(\delta^*(q,u),a)$, where $u \in \Sigma^*$, $a \in \Sigma$, and $q \in Q$. The behavior of a DFA A can also be described by so called *configurations*. For $q \in Q$ and $u \in \Sigma^*$, the configuration $q \cdot u$ mirrors the current status of the automaton, that is, A is in state q, and u is the unprocessed part of the input word. By $\vdash_A$ we denote the *next-step relation*, which is defined as follows: if $\delta(q,a) = p$ and A is in the configuration $q \cdot au$, then A performs the computation step $q \cdot au \vdash_A p \cdot u$. Here $\vdash_A^*$ denotes the reflexive and transitive closure of $\vdash_A$. Finally, a finite state automaton is called *non-deterministic* (NFA for short) if δ is a relation, that is, a function from $Q \times \Sigma$ onto $\mathfrak{P}(Q)$.[1] The language computed by A is defined as $L(A) = \{w \in \Sigma^* \mid q_0 \cdot w \vdash_A^* q \cdot \varepsilon \text{ and } q \in F\}$. It is well known that the language classes characterized by DFAs and NFAs coincide.

For CFL the corresponding device is a pushdown automaton. A *pushdown automaton* $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ (PDA for short) is a 7-tuple, where everything is described as for finite state automata. Additionally, Γ is the finite stack alphabet, and Z_0 is the initial stack symbol. Finally δ is defined as

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathfrak{P}_{fin}(Q \times \Gamma^*).$$

A *configuration* of a PDA $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a tuple $(q \cdot u, \gamma)$, where $q \cdot u$ is described as for finite state automata and γ is the current stack content. Accordingly, the next step relation $\vdash_A$ easily extends to PDA. For that note that the stack is used as follows: in every step A reads the last symbol on the stack and replaces it by a string over Γ. The language accepted by A is $L(A) = \{w \in \Sigma^* \mid (q_0 \cdot w, Z_0) \vdash_A^* (q \cdot \varepsilon, \alpha) \text{ and } q \in F,\ \alpha \in \Gamma^*\}$. Note that here we omit to define a second acceptance criterion, the acceptance by empty stack, as both ways are equivalent. Again a pushdown automaton is called *deterministic* (DPDA for short) if for all $q \in Q$, $\alpha \in \Gamma$ and $a \in \Sigma$, $|\delta(q,\varepsilon,\alpha)| + |\delta(q,a,\alpha)| \leq 1$ holds. Observe that the DPDAs form a subclass of the PDAs, which accept the deterministic context-free languages, denoted by DCFL.

Since for us characterizations of the classes CSL and RE by automata play a minor role, we only provide little information on these types of machines here. A *(non-deterministic) Turing Machine* $A = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ (TM for short) is a 7-tuple, where everything is described as for finite state automata. Additionally, Γ is the finite tape alphabet with

[1] Note that there are several variants of finite state automata that are e.g. capable to use several initial states or arbitrary ε-steps.

$\Sigma \subsetneq \Gamma$ and $\square \in \Gamma$ is the blank symbol. Finally, δ is defined as

$$\delta : Q \times \Gamma \to \mathfrak{P}(Q \times \Gamma \times \{L, R, N\}).$$

The notion of configuration, computation, and accepted language extends again from the previously introduced automata. Informally, a Turing Machine differs from the automata above, by working on an infinite tape, by moving the head in both directions, and by rewriting symbols. While entering a final state, the machine accepts the input word. The class of languages accepted by Turing Machines is exactly the class RE.

Finally we call a Turing Machine linear bounded (LBA for short) if the size of its tape is bounded by the length of the input word. LBAs are the automata equivalent to the class CSL. According to the previously introduced automata, we may also define a deterministic version of a linear bounded Turing Machine. This machine accepts the class of deterministic context-sensitive languages (DCSL for short). Note that the equivalence of CSL and DCSL is a longstanding open question in computer science theory, known as the LBA-Problem.

In conclusion the following strict inclusions can be obtained for the classes introduced above. This chain is known as the Chomsky Hierarchy.

Theorem 2.1.4. $\mathsf{REG} \subset \mathsf{DCFL} \subset \mathsf{CFL} \subset \mathsf{CSL} \subset \mathsf{RE}$.

2.2 Restarting Automata

Since restarting automata were introduced in 1995 by Jančar, Mráz, Plátek and Vogel [JMPV95], they became a vivid research topic, which lead to numerous remarkable approaches in the field of Formal Language Theory. Here we outline those results, which have influence on the present work and provide some new findings for restarting automata. For that we start with the motivation and some formal definitions. However, a nice survey of restarting automata can be found in [Ott06], which also serves as the main reference for the present section[2].

Analysis by Reduction

As mentioned in the introduction a restarting automaton mirrors the linguistic technique of Analysis by Reduction. Hence, a closer look on this technique is mandatory for under-

[2] Additionally some phrases have been taken from the introductions of [Mes08, Sta08].

standing the motivation of combining restarting automata and binary word relations in the following chapters.

Analysis by Reduction is simply a method to verify the (syntactical) correctness of a given sentence in natural languages. It is recommended for "free word order" languages, such as many Slavic languages (e.g. Czech, Sorbian, Russian, etc.). Accordingly, a given sentence is simplified stepwise under the condition that every step preserves its correctness or incorrectness. The principles of this technique can be illustrated by the following example, taken from [Ott06].

They mean that the means she means are very mean.

Analysis by Reduction starts with reading the sentence from left to right until a phrase is discovered that can be simplified. For example the word *that* or the word *very* can be deleted:

They mean the means she means are very mean.

They mean that the means she means are mean.

Obviously both simplifications are correct. Hence, *that* and *very* are independent of each other. In summary, the following sentence is derived by sequentially applying the last two steps.

They mean the means she means are mean.

Next, it is easy to observe that *the means* is not deletable, as this would result in an incorrect sentence. The reason for which is that the phrase *They mean* depends on *the means*. However, we can conclude that *They mean* and *she means* are independent of each other and obsolete for the syntactical correctness of the sentence. Then, the simple sentence

The means are mean.

is said to be a basis form of the given input sentence, which is easily verified as being (syntactically) correct. In addition, we have seen that some information about the dependency structure of the sentence can be obtained.

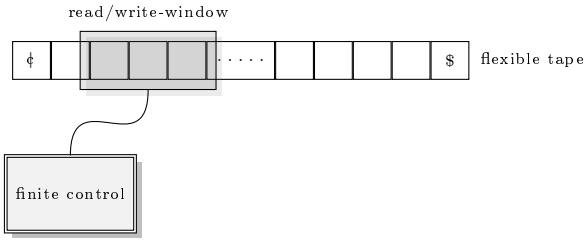

Figure 2.1: Schematic representation of a restarting automaton.

The previous example immediately gives some insights in the process of natural language analyzation. For that we briefly mention some facts here, which are motivating for reflections in the following chapters.

It is easy to observe that the example sentence is highly ambiguous, as the word *mean* intends three different meanings. Thus, when it comes to a non-human driven Analysis by Reduction, ambiguities have to be resolved. Without going into details, the latter might be done by adding information about linguistic categories (i.e. morphology, syntax, etc.) to each word. For instance, the *object means*, which is the *plural form* of mean, is a *noun*. Hence, actually Analysis by Reduction is applied on sentences enriched with auxiliary symbols that are used to disambiguate the given input. Here, we recommend [LPS07] for further reading on the role of Analysis by Reduction (and therewith the role of restarting automata) in the language analyzation process.

Definition and Examples

Since Analysis by Reduction intends a cycle-wise reduction of a given sentence of a natural language, this can be schematically depicted as shown in Figure 2.1. Hence, a restarting automaton consists of a finite-state control, a flexible tape with end markers, and a read/write window of a fixed size working on that tape. It works in cycles, where in each cycle it performs a single rewrite operation that shortens the tape contents. Every cycle ends with a restart operation that forces the automaton to reset the internal state to the initial one. After a finite number of cycles, it halts and accepts (or rejects) the input. That is, restarting automata are obviously language accepting devices.

Definition 2.2.1. *A restarting automaton (*RRWW *for short) is defined as an 8-tuple* $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$*, where* Q *is the finite set of states,* Σ *and* Γ *are the finite input*

and tape alphabet, $¢, \$ \notin \Gamma$ *are the markers for the left and right border of the tape,* $q_0 \in Q$ *is the initial state, and* $k \geq 1$ *is the size of the read/write window. Additionally, the transition function* δ *is defined as*

$$\delta : Q \times \mathcal{PC}^{(k)} \to \mathfrak{P}(Q \times (\{\mathsf{MVR}\} \cup \mathcal{PC}^{\leq(k-1)}) \cup \{\mathsf{Restart}, \mathsf{Accept}\}),$$

where $\mathcal{PC}^{(k)}$ *denotes the set of possible contents of the read/write window of* M*, that is,*

$$\mathcal{PC}^{(i)} = (¢ \cdot \Gamma^{i-1}) \cup \Gamma^i \cup (\Gamma^{\leq i-1} \cdot \$) \cup (¢ \cdot \Gamma^{\leq i-2} \cdot \$) \quad (i \geq 0)$$

and

$$\Gamma^{\leq n} = \bigcup_{i=0}^{n} \Gamma^i \quad \textit{and} \quad \mathcal{PC}^{\leq(k-1)} = \bigcup_{i=0}^{k-1} \mathcal{PC}^{(i)}.$$

The transition function δ of a restarting automaton describes five different types of transition steps:

1. A *move-right step* is of the form $(q', \mathsf{MVR}) \in \delta(q, u)$, where $q, q' \in Q$ and $u \in \mathcal{PC}^{(k)}$, $u \neq \$$. If M is in state q and sees the string u in its read/write window, then this step causes M to shift the read/write window one position to the right and to enter state q'. However, if the content u of the read/write window is only the symbol \$, then no shift to the right is possible.

2. A *rewrite step* is of the form $(q', v) \in \delta(q, u)$, where $q, q' \in Q$, $u \in \mathcal{PC}^{(k)}$, $u \neq \$$ and $v \in \mathcal{PC}^{\leq(k-1)}$ such that $|v| < |u|$. It causes M to replace the content u of the read/write window by the string v, thereby shortening the tape and to enter state q'. Further, the read/write window is placed immediately to the right of the string v. However, some additional restrictions apply in that the border markers ¢ and \$ must not disappear from the tape nor that new occurrences of these markers are created. Finally, the read/write window must not move across the right border marker \$, that is, if the string u ends in \$, then so does the string v. After performing the rewrite operation, the read/write window is placed on the \$-symbol.

3. A *restart step* is of the form $\mathsf{Restart} \in \delta(q, u)$, where $q \in Q$ and $u \in \mathcal{PC}^{(k)}$. It causes M to place its read/write window over the left end of the tape, so that the first symbol it sees is the left border marker ¢ and to reenter the initial state q_0.

4. An *accept step* is of the form $\mathsf{Accept} \in \delta(q, u)$, where $q \in Q$ and $u \in \mathcal{PC}^{(k)}$. It causes M to halt and accept.

If $\delta(q, u) = \emptyset$ for some $q \in Q$ and $u \in \mathcal{PC}^{(k)}$, the M necessarily halts, and we say that M *rejects* in this situation. Further, the letters in $\Gamma \backslash \Sigma$ are called *auxiliary symbols*.

A *configuration* of M is described by a string $\alpha q \beta$, where $q \in Q$, and either $\alpha = \varepsilon$ and $\beta \in \{¢\} \cdot \Gamma^* \cdot \{\$\}$ or $\alpha \in \{¢\} \cdot \Gamma^*$ and $\beta \in \Gamma^* \cdot \{\$\}$. Here q represents the current state, $\alpha\beta$ is the current content of the tape, where the read/write window contains the first k symbols of β or all of β when $|\beta| \leq k$. The three most important situations that may occur during a computation of the restarting automaton M are described by the following configurations. At the beginning of a computation M is in the *initial configuration* $q_0 ¢ w \$$, where $w \in \Sigma^*$ is the input word. A *restarting configuration* is described by $q_0 ¢ w' \$$, where $w' \in \Gamma^*$. Observe that the initial configuration is a particular type of restarting configuration. Further, we use Accept to denote the *accepting configuration*s, which are those configurations that M reaches by performing an accept step. A configuration of the form $\alpha q \beta$, such that $\delta(q, \beta') = \emptyset$, where β' is the current content of the read/write window, is a *rejecting configuration*. Finally, a *halting configuration* is either an accepting or a rejecting configuration.

In general, the automaton M is *non-deterministic*, that is, there can be two or more instructions with the same left-hand side $\delta(q, u)$. Thus, there can be more than one computation for an input word. Obviously, if this is not the case, then the automaton is deterministic. Here we omit further details on possible variants of the "standard" model. However, they are summarized in the following subsection. We continue by taking a closer look on the mode of operation of the restarting automaton.

Observe that any finite computation of the restarting automaton M consists of certain phases. Such a phase is called a *cycle*, where each cycle is a combination of a number of move-right steps (MVR), exactly one rewrite step, and a restart- ($\mathsf{Restart}$) or accept step (Accept). Hence, a cycle of M starts in a restarting configuration, the head moves along the tape until somewhere a rewrite instruction can be applied. Then the head continues moving until a restart operation is performed and thus a new restarting configuration is reached. If no further restart operation is performed, any finite computation finishes in a halting configuration. Such a phase is called a *tail*. Note that during a tail also at most one rewrite operation may be executed.

Accordingly, an accepting computation of M consists of a finite sequence of cycles that is

followed by an accepting tail computation. It can be described as

$$q_0\¢ w\$ \vdash_M^c q_0\¢ w_1\$ \vdash_M^c \ldots \vdash_M^c q_0\¢ w_m\$ \vdash_M^* \mathsf{Accept},$$

where $w \in \Sigma^*$ denotes the input word of M and $w_1, ..., w_n \in \Gamma^*$ are the shorter tape contents that occur during the computation. Further $\vdash_M$ denotes the single step relation and $\vdash_M^c$ denotes the execution of a complete cycle. Finally $\vdash_M^*$ and $\vdash_M^{c^*}$ are the reflexive and transitive closures of these relations.[3] Hence, an input $w \in \Sigma^*$ is accepted by M, if there exists a computation which starts with the initial configuration on input w and ends in the accepting configuration. Then, M accepts the following language:

$$L(M) = \{w \in \Sigma^* \mid q_0\¢ w\$ \vdash_M^* \mathsf{Accept}\}.$$

We illustrate the above definitions by the following example. The shown restarting automaton is taken from [Ott06], but it is slightly adapted to mirror all capabilities that are implied by our definition of restarting automata.

Example 2.2.2. Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, 4, \delta)$ be the RRWW-automaton that is defined by taking $Q = \{q_0, q_c, q_d, q_r\}$, $\Sigma = \{a, b, c, d\}$, $\Gamma = \{a, b, c, d, \hat{c}, \hat{d}\}$. Further, δ is given by the following transition table:

$$\begin{array}{rrcl}
(1) & \delta(q_0, ¢ax) & = & (q_0, \mathsf{MVR})\ (x \in \{aa, ab, bb, bc\}), \\
(2) & \delta(q_0, aax) & = & (q_0, \mathsf{MVR})\ (x \in \{aa, ab, bb), \\
(3) & \delta(q_0, abbb) & = & \{(q_c, \hat{c}bb), (q_d, \hat{d}b)\}, \\
(4) & \delta(q_0, abbd) & = & (q_d, \hat{d}d), \\
(5) & \delta(q_0, abc\$) & = & (q_c, \hat{c}c\$), \\
(6) & \delta(q_c, bbbb) & = & (q_c, \mathsf{MVR}), \\
(7) & \delta(q_c, bbbc) & = & (q_c, \mathsf{MVR}), \\
(8) & \delta(q_c, bbc\$) & = & \mathsf{Restart}, \\
(9) & \delta(q_d, bbbb) & = & (q_d, \mathsf{MVR}), \\
(10) & \delta(q_d, bbbd) & = & (q_d, \mathsf{MVR}), \\
(11) & \delta(q_d, bbd\$) & = & \mathsf{Restart}, \\
(12) & \delta(q_0, ¢ax) & = & (q_0, \mathsf{MVR})\ (x \in \{a\hat{c}, a\hat{d}, \hat{c}b, \hat{d}b\}), \\
(13) & \delta(q_0, aax) & = & (q_0, \mathsf{MVR})\ (x \in \{a\hat{c}, a\hat{d}, \hat{c}b, \hat{d}b\}),
\end{array}$$

[3] Note that we often use the single step relation to express what kind of step is performed. Thus, $\vdash_{\mathsf{MVR}}$ or $\vdash^{\mathsf{MVR}}$ denotes a move-right step.

$$
\begin{array}{lll}
(14) & \delta(q_0, a\hat{c}bb) & = (q_r, \hat{c}b), \\
(15) & \delta(q_0, a\hat{c}bc) & = (q_r, \hat{c}c), \\
(16) & \delta(q_0, a\hat{d}bb) & = (q_r, \hat{d}), \\
(17) & \delta(q_r, x) & = \mathsf{Restart}\ (x \in \Sigma^* \cup \{\$\}), \\
(18) & \delta(q_0, \mathfrak{c}c\$) & = \mathsf{Accept}, \\
(19) & \delta(q_0, \mathfrak{c}d\$) & = \mathsf{Accept}, \\
(20) & \delta(q_0, \mathfrak{c}\hat{c}c\$) & = \mathsf{Accept}, \\
(21) & \delta(q_0, \mathfrak{c}\hat{d}d\$) & = \mathsf{Accept}.
\end{array}
$$

We will show next that M accepts the language

$$L = \{a^n b^n c \mid n \geq 0\} \cup \{a^n b^{2n} d \mid n \geq 0\}.$$

From the transitions (18) and (19) it is clear that M accepts c and d immediately. So let $w \in \Sigma^+ \backslash \{c, d\}$. Starting from the initial configuration $q_0 \mathfrak{c} w \$$, M will get stuck (and therewith reject) while scanning a prefix of w unless this prefix is of the form $a^n b$ ((1) and (2)) for some positive integer n. In this case the configuration $\mathfrak{c} a^{n-1} q_0 aby\$$ is reached, where $y \in \Sigma^+$. Depending on y, M continues with (3), (4) or (5), that is, it either rewrites a factor ab to $\hat{c}$ or abb to $\hat{d}$. Here we only describe the behavior of M for sufficiently long inputs, since it is clear from the transition function how the restarting automaton behaves if $|w| \leq 4$. Thus, M guesses if the word ends with c or d. This guess is saved while writing the auxiliary symbols $\hat{c}$ (respectively $\hat{d}$) on the tape. Now M is in one of the two configurations $\mathfrak{c} a^{n-1} \hat{c} bbq_c y'\$$ or $\mathfrak{c} a^{n-1} \hat{d} bq_d y'\$$. In the first case the transitions (6), (7) and (8) ((9), (10) and (11)) are used to scan the suffix y' of w and restarting if c (respectively d) is the last symbol on the tape. From here on the restarting automaton acts deterministically, that is, in the following cycles M deletes a factor ab or abb according to the written auxiliary symbol ((12) - (16)). Finally M accepts while seeing either $\hat{c}c$ or $\hat{d}d$ on its tape ((20) and (21)). Thus it is clear that $L(M) = L$.

Basic Properties

Next we restate some basic facts about computations of restarting automata.

Proposition 2.2.3 (Error Preserving Property for Restarting Automata)**.** *Let $M = (Q, \Sigma, \Gamma, \mathfrak{c}, \$, q_0, k, \delta)$ be an* RRWW*-automaton, and let u and u' be words over its input alphabet Σ. If $q_0 \mathfrak{c} u\$ \vdash_M^{c^*} q_0 \mathfrak{c} u'\$$ holds and $u \notin L(M)$, then $u' \notin L(M)$, either.*

Proposition 2.2.4 (Correctness Preserving Property for Restarting Automata). *Let* $M = (Q, \Sigma, \Gamma, \mathfrak{c}, \$, q_0, k, \delta)$ *be an* RRWW*-automaton, and let* u *and* u' *be words over its input alphabet* Σ*. If* $q_0\mathfrak{c}u\$ \vdash_M^{c^*} q_0\mathfrak{c}u'\$$ *is an initial segment of an accepting computation of* M*, then* $u' \in L(M)$*.*

The following property leads to a simple way of describing the behavior of restarting automata.

Lemma 2.2.5. *Each* RRWW*-automaton is equivalent to an* RRWW*-automaton that makes an accept or restart step only when it sees the right border marker* \$ *in its read/write window.*

This lemma means that in each cycle of each computation and also during the tail of each computation the read/write window moves all the way to the right before a restart is made, respectively, before the machine halts and accepts.

Based on this fact each cycle (and also the tail) of a computation of an RRWW-automaton can be described through a sequence of so-called *meta-instructions* [NO01] of the form $(E_1, u \to u', E_2)$, where E_1 and E_2 are regular languages, called *regular constraints* of this instruction, and u and u' are strings such that $|u| > |u'|$. The rule $u \to u'$ stands for a rewrite step of the RRWW-automaton. On trying to execute this meta-instruction an RRWW-automaton will get stuck (and so reject) starting from the configuration $q_0\mathfrak{c}w\$$, if w does not admit a factorization of the form $w = w_1uw_2$ such that $\mathfrak{c}w_1 \in E_1$ and $w_2\$ \in E_2$. On the other hand, if w does have a factorization of this form, then one such factorization is chosen non-deterministically, and $q_0\mathfrak{c}w\$$ is transformed into $q_0\mathfrak{c}w_1u'w_2\$$. In order to describe the tails of accepting computations we use meta-instructions of the form $(\mathfrak{c} \cdot E \cdot \$, \mathsf{Accept})$, where the strings from the regular language E are accepted by the RRWW-automaton in tail computations. We illustrate this concept by describing the RRWW-automaton from Example 2.2.2 by meta-instructions.

Example 2.2.6. Again, let $M = (Q, \Sigma, \Gamma, \mathfrak{c}, \$, q_0, 4, \delta)$ be the RRWW-automaton that is defined by taking $Q = \{q_0, q_c, q_d, q_r\}$, $\Sigma = \{a, b, c, d\}$, $\Gamma = \{a, b, c, d, \hat{c}, \hat{d}\}$. Instead of defining δ directly, we give the following sequence of meta-instructions for M:

$$
\begin{array}{lllll}
(1) & (\mathfrak{c} \cdot a^*, & ab \to \hat{c}, & b^* \cdot c \cdot \$), \\
(2) & (\mathfrak{c} \cdot a^*, & abb \to \hat{d}, & b^* \cdot d \cdot \$), \\
(3) & (\mathfrak{c} \cdot a^*, & a\hat{c}b \to \hat{c}, & \Sigma^* \cdot \$), \\
(4) & (\mathfrak{c} \cdot a^*, & a\hat{d}bb \to \hat{d}, & \Sigma^* \cdot \$), \\
(5) & (\mathfrak{c} \cdot \{c, d\} \cdot \$, & \mathsf{Accept}), & \\
(6) & (\mathfrak{c} \cdot \hat{c}c \cdot \$, & \mathsf{Accept}), & \\
(7) & (\mathfrak{c} \cdot \hat{d}d \cdot \$, & \mathsf{Accept}). &
\end{array}
$$

It is easily seen that this set of meta-instructions can be transformed into the transition function δ from Example 2.2.2 and thus $L(M) = \{a^n b^n c \mid n \geq 0\} \cup \{a^n b^{2n} d \mid n \geq 0\}$.

This way of describing RRWW-automaton will become quite important throughout the fifth chapter of this work as in [NO00] it was shown that this corresponds to the characterization of the class $\mathcal{L}$(RRWW) by certain infinite prefix-rewriting systems.

Variants of Restarting Automata

In recent years a lot of different restrictions and extensions of standard[4] restarting automata were discussed and investigated. Here we present a selection of these modifications, in fact, mainly those that are useful for our purposes.

Let us start with restrictions on the rewrite and move process. These restrictions are expressed by the number of R's and W's in the description of the restarting automaton. Here we distinguish between six types, that are, RRWW-, RWW-, RW-, R-, RRW- and RR-automata. The meaning of these R's and W's can be summarized as follows:

one R **[RWW-, RW- and R-automata]** These types of restarting automata differ from the standard model in that they have to restart immediately after a rewrite operation. In particular, this means that they cannot perform a rewrite step during the tail of a computation.

one W **[RW- and RRW-automata]** Restarting automata of these types are not allowed to use auxiliary symbols, that is, their tape alphabets coincide with the input alphabets.

[4] Based on Definition 2.2.1.

no W **[RR- and R-automata]** These types of restarting automata can only delete symbols. Hence, the right hand side u' of each rewrite step $(q', u') \in \delta(q, u)$ is a scattered sub-word of the left-hand side u.

Next we present two further restrictions on the computation of restarting automata. First of all, the prefix det- denotes *deterministic* restarting automata, that is, $|\delta(q, u)| \leq 1$ for all states q and all possible window contents u.

Secondly, the prefix mon- denotes *monotone* restarting automata. The notion of monotonicity was first introduced in [JMPV97]. Here we will use a slightly generalized definition taken from [JMOP06]. Let M be an RRWW-automaton. Each computation of M can be described by a sequence of cycles $C_1, C_2, \ldots, C_n$, where C_n is the last cycle, which is followed by the tail of the computation. Each cycle C_i of this computation contains a unique configuration of the form $\mathfrak{c}xquy\$$ such that q is a state and $(q', u') \in \delta(q, u)$ is the rewrite step that is applied during this cycle. By $D_r(C_i)$ we denote the *right distance* $|uy\$|$ of this cycle. The sequence of cycles $C_1, C_2, \ldots, C_n$ is called monotone if $D_r(C_1) \geq D_r(C_2) \geq \cdots \geq D_r(C_n)$ holds. A computation of M is called monotone if the corresponding sequence of cycles is monotone. Observe that the tail of the computation is not taken into account here. Finally, the RRWW-automaton M is called monotone if each of its computations that starts from an initial configuration is monotone.

Finally, we consider an extension of restarting automata, which was originally defined in [MS04]. There restarting automata were introduced that must not reset their internal state to the initial state while performing a restart operation. Hence, this ability can be used to carry over some information from one cycle to the next. Therefore, restarting automata of that type are called *non-forgetting*, which is denoted by the prefix nf-. Observe that in this case the notion of meta-instructions easily carries over to non-forgetting restarting automata. Thus, a meta-instruction of a non-forgetting restarting automaton is described as a 5 tuple $(q, E_1, u \to u', E_2, q')$, where E_1, E_2 and $u \to u'$ are defined as before and q denotes the restart state of the described cycle and q' is the state the automaton must restart in after completing this cycle[5].

In the following we summarize the main results concerning the different types of restarting automata stated above.

[5] In contrast to the situation for meta-instructions we use various ways of denoting restarting transitions throughout the text. This will increase the readability of technical details in the respective context. However, it will be clear when a particular transition denotes a restarting transition.

General Classification

On a given input of length n, clearly an RRWW-automaton can execute at most n cycles. Thus, the following upper bounds for the computational power of restarting automata can immediately be established, where P and NP denote the well-known complexity classes.

Proposition 2.2.7.

$$\begin{array}{lllll} (a) & \mathcal{L}(\mathsf{RRWW}) & \subseteq & \mathsf{NP} \cap \mathsf{CSL} \\ (b) & \mathcal{L}(\mathsf{det\text{-}RRWW}) & \subseteq & \mathsf{P} \cap \mathsf{DCSL} \end{array}$$

Obviously each type of restarting automaton is an extension of a finite state automaton. Therefore the regular languages (REG) form a lower bound.

However, Figure 2.2 summarizes the relations between the various types of restarting automata and well-known language classes. Here, arrows denote proper inclusions and if there is no connection, then the classes are incomparable. Further, by GCSL we denote the well-known class of growing context-sensitive languages introduced in [DW86], which is defined by strictly monotone grammars. And CRL denotes the class of Church-Rosser languages, exposed in [MNO88]. Finally it is worth to mention that it is still open whether the inclusion $\mathcal{L}(\mathsf{RWW}) \subseteq \mathcal{L}(\mathsf{RRWW})$ is proper, and whether $\mathcal{L}(\mathsf{RRW})$ is contained in $\mathcal{L}(\mathsf{RWW})$. Both are in fact longstanding open questions (e.g. posed in [JMPV98]).

Monotone Restarting Automata

The notion of monotonicity leads to some quite interesting characterizations in terms of restarting automata. In [JMPV97, JMPV98, JMPV99] it was shown that all language classes accepted by deterministic restarting automata that are monotone coincide with the deterministic context-free languages (DCFL). Further, the non-deterministic versions form a hierarchy inside the context-free languages (CFL). Here it is also worth to notice that the property of being monotone is decidable for any RRWW-automaton [Ott]. The mentioned results are summarized in Figure 2.3.

Next we add the non-forgetting (nf-) property to the classes defined in this section. Then, in contrast to the results above, not all deterministic classes coincide [MO11]. In particular, the property of scanning the rest of the tape (that means two R's in the description of the restarting automaton) makes a difference while increasing the power beyond DCFL. Here most noteable is the characterization of the class of *left-to-right regular languages*[6] (LRR

[6] The class LRR was introduced in [IC73].

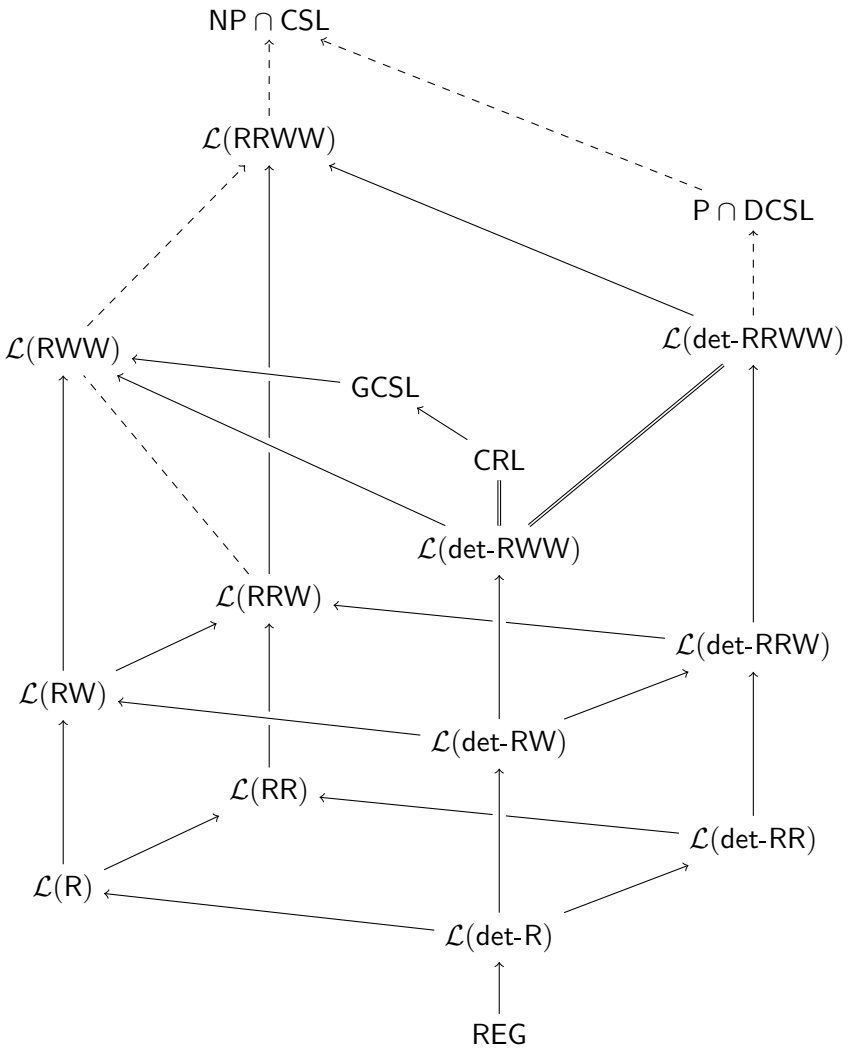

Figure 2.2: Inclusions between types of restarting automata and well-known language classes. Proper inclusions are denoted by arrows, inclusions not known to be proper by dashed arrows, and unknown relationships by dashed lines.

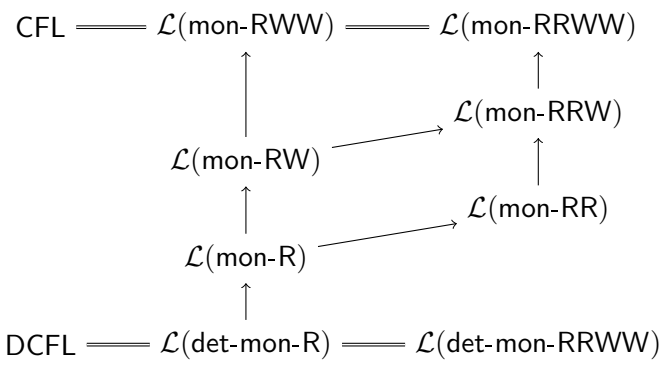

Figure 2.3: Inclusions between the various types of monotone restarting automata.

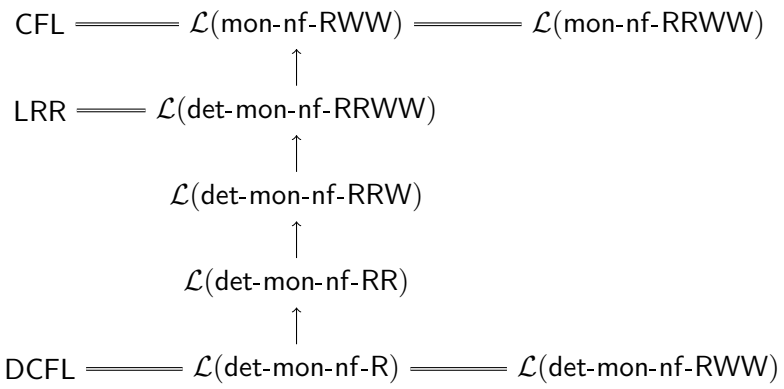

Figure 2.4: Hierarchy of some important classes of non-forgetting restarting automata.

for short) by det-mon-nf-RRWW-automata. Further, the non-deterministic classes form exactly the same hierarchy as the forgetting variants [Mes08]. To this end we again summarize the most important hierarchy results taken from [MO11] and [Mes08] in Figure 2.4.

2.2.1 Restarting Automata with Window Size One

One of the parameters that are essential for a restarting automaton is the size of the read/write window. Mráz showed in [Mrá01] that for automata without auxiliary symbols an increase of the size of the read/write window increases also the power of these automata. Further, it was shown that in this context the regular languages always form a lower bound. That is, if the size of the read/write window is restricted to 1, most types of restarting automata characterize the regular languages. In particular, the following results have been obtained.

Proposition 2.2.8 ([Mrá01])**.**

$$\mathcal{L}(\text{det-R}(1)) = \mathcal{L}(\text{det-mon-R}(1)) = \mathcal{L}(\text{mon-R}(1)) = \mathcal{L}(\text{R}(1)) = \text{REG}.$$

Reimann ([Rei07]) extended the previous results to det-RR(1)-automata.

Proposition 2.2.9 ([Rei07])**.**

$$\mathcal{L}(\text{det-mon-RR}(1)) = \mathcal{L}(\text{det-RR}(1)) = \text{REG}.$$

In fact, the authors above have only shown that the language class computed by R(1)- respectively det-RR(1)-automata coincides with the regular languages. Since each type of restarting automaton can compute the regular languages, and deterministic restarting automata with window size one are necessarily monotone, also the additional equivalences given in Proposition 2.2.8 and Proposition 2.2.9 hold. Indeed, the latter fact can easily be verified. Let $M = (Q, \Sigma, ¢, \$, q_0, 1, \delta)$[7] be a deterministic RR(1)-automaton, and assume that M executes the cycle $q_0¢w\$ = q_0¢uav\$ \vdash_M^c q_0¢uv\$$. Then starting from the configuration $q_0¢uv\$$, M must move right across the prefix $¢u$ of the tape contents before it can possibly detect an applicable rewrite operation, as M is deterministic. Hence, if the

[7]Note that restarting automata with window size one simply erase a single symbol from the tape in each cycle. Hence, to avoid redundancy we describe them as 7-tuples, for the reason that in this case the tape alphabet always coincides with the input alphabet.

computation continues with the cycle $q_0\¢uv\$ = q_0\¢xby\$ \vdash_M^c q_0\¢xy\$$, then u is a prefix of x, that is, the right distance of the second cycle is smaller than the right distance of the former cycle. It follows that M is monotone.

Further, it is worth to notice that already mon-RR(1)-automata can compute non-regular languages. For that consider the following example, taken from [KO12].

Example 2.2.10. Let $M = (Q, \{a, b\}, ¢, \$, q_0, 1, \delta)$ be the mon-RR(1)-automaton that is described by meta-instructions as follow:

$$\begin{array}{lllll}
(1) & (¢ \cdot (aa)^* \cdot a, & a \to \varepsilon, & (bb)^* \cdot \$), \\
(2) & (¢ \cdot (aa)^* \cdot a, & b \to \varepsilon, & b \cdot (bb)^* \cdot \$), \\
(3) & (¢ \cdot (aa)^*, & a \to \varepsilon, & b \cdot (bb)^* \cdot \$), \\
(4) & (¢ \cdot (aa)^*, & b \to \varepsilon, & (bb)^* \cdot \$), \\
(5) & (¢ \cdot \$, & \mathsf{Accept}). &
\end{array}$$

Clearly the automaton M processes inputs of the form $a^m b^n$, and it can be easily verified that M is monotone and $L(M) = \{a^n b^n, a^n b^{n+1} \mid n \geq 0\}$. For that just observe that it removes the first occurrence of the letter b and the last occurrence of the letter a, alternatingly. To distinguish between these two cases it uses the parity of the number of a's and the parity of the number of b's. For instance, M deletes an a if and only if the number of a's and b's are both even or odd. Clearly b's are treated in the opposite way.

We continue by extending the above characterizations of REG to non-forgetting restarting automata. We will see in Chapter 4 that the following results open up a quite interesting and fruitful topic, namely relations that are defined by transducers of that type. Further, the upcoming results are also fairly surprising from a language theoretic point of view, for the reason that the property of being non-forgetting generally increases the power of restarting automata.

Theorem 2.2.11. $\mathcal{L}(\text{det-mon-nf-R}(1)) = \text{REG}$.

Proof. We mentioned already that every type of restarting automaton can accept the regular languages just by simulating a DFA in its finite control. Thus, it remains to prove the inclusion from left to right. Let $M = (Q, \Sigma, ¢, \$, q_0, 1, \delta)$ be a det-mon-nf-R(1)-automaton for $L \subseteq \Sigma^*$. W.l.o.g. we may assume that $Q = \{q_0, q_1, \ldots, q_{n-1}\}$, and that M executes accept-instructions only on reading the \$-symbol. Below we describe a deterministic finite-state acceptor (DFA) A for the language $L \cdot \$$. In order to illustrate the

main problem that we must overcome in simulating a monotone non-forgetting restarting automaton by a finite-state acceptor, we begin by looking at an example computation of M.

If $w \in L$, then the computation of M on input w is accepting. Let us assume that this computation consists of a sequence of at least two cycles that is followed by a tail computation, that is, it has the following form:

$$\begin{array}{rcllll} q_0\mathbin{¢}w\$ & = & q_0\mathbin{¢}uav\$ & \vdash^+_{\mathsf{MVR}} & \mathbin{¢}up_0av\$ & \vdash_M \; q_{i_1}\mathbin{¢}uv\$ \\ & = & q_{i_1}\mathbin{¢}xby\$ & \vdash^+_{\mathsf{MVR}} & \mathbin{¢}xp_{i_1}by\$ & \vdash_M \; q_{i_2}\mathbin{¢}xy\$ \\ & \vdash^{c^*}_M & q_{i_m}\mathbin{¢}z\$ & \vdash^+_{\mathsf{MVR}} & \mathbin{¢}zp_{i_m}\$ & \vdash_M \; \mathsf{Accept}, \end{array}$$

where $u, v, x, y, z \in \Sigma^*$, $a, b \in \Sigma$, and $p_0, q_{i_1}, p_{i_1}, q_{i_2}, q_{i_m}, p_{i_m} \in Q$. The right distance of the first cycle is $d_1 = |v| + 2$, and the right distance of the second cycle is $d_2 = |y| + 2$. As M is monotone, we have $d_1 \geq d_2$, that is, $|v| \geq |y|$. Since $uv = xby$, this means that y is a suffix of v.

If $d_1 > d_2$, that is, the sequence of these two cycles is *strictly monotone*, then y is a proper suffix of v, and so $v = x_2by$ for a suffix x_2 of x. In this case $w = uav = uax_2by$, which implies that the second delete operation is executed at a place that is strictly to the right of the place where the first delete operation was executed. In this situation the DFA A will first encounter the letter a deleted in the first cycle, and later it will encounter the letter b deleted in the second cycle.

If, however, $d_1 = d_2$, that is, the sequence of two cycles considered is *not* strictly monotone, then $v = y$, and so $uv = xby = xbv$ implies that $u = xb$, which yields $w = uav = xbav$. Thus, in this situation the second delete operation is executed immediately to the *left* of the place where the first delete operation was executed. Hence, in this case we have the problem that the DFA A will first encounter the letter b that is deleted in the second cycle *before* it will encounter the letter a that is deleted in the first cycle. Somehow we must overcome this problem.

Notice that the latter situation described above can happen only if M completes the first cycle by restarting in the correct state q_{i_1}. In fact, this situation can occur more than once in a row. However, as M is deterministic, we see that $q_{i_1} \neq q_0$, and more generally, if $C_1, C_2, \ldots, C_k$ is a sequence of cycles such that $D_r(C_1) = \cdots = D_r(C_k)$, then all these cycles must begin in different states. This implies that the length of such a sequence of cycles is bounded from above by the number n of states of M.

In order to simulate the computation of M on input w, the DFA A must store some information on the computation of M it tries to simulate in its finite-state control. In particular, it needs to store information related to sequences of cycles of M that are not strictly monotone. For this we assume that as a part of its internal states A stores the following data structures:

- The *current restart state* CRS that contains the state $q \in Q$ with which the cycle of M started that is currently active. Initially CRS is set to q_0.
- A *state table* T that initially contains the list of all pairs (q_i, q_i') $(0 \leq i \leq n-1)$, where q_i' is the state that M enters from state q_i on seeing the $\mathfrak{c}$-symbol. If $\delta(q_i, \mathfrak{c})$ is undefined, then T contains the item $(q_i, -)$.
- A *buffer* B of length at most n that is initially empty. It will be used to store information on possible rewrite (that is, delete) operations encountered during the current simulation.

When simulating the above computation of M, the DFA A will store the following information:

- CRS still contains the initial state q_0;
- the table T contains the pairs (q_i, p_i) $(0 \leq i \leq n-1)$, where p_i is the state that M reaches from the restarting configuration $q_i\mathfrak{c}w\$ = q_i\mathfrak{c}xbav\$$ by moving right across the prefix $\mathfrak{c}x$;
- on realizing that M can execute the rewrite operation $\delta(p_{i_1}, b) = (q_{i_2}, \varepsilon)$, A stores the pair $(b, (q_{i_1}, p_{i_1}, q_{i_2}))$ in B, and moves right to the next letter. In fact, such a pair is stored in B for each index j such that $\delta(p_j, b)$ is a rewrite operation. Together with these pairs, A also stores a copy of the current state table T. Recall that, for all i, T contains the pair (q_i, p_i) such that, when starting in state q_i, M reaches state p_i after reading across the prefix $\mathfrak{c}x$.

Next A realizes that M can execute the rewrite operation $\delta(p_0, a) = (q_{i_1}, \varepsilon)$. Hence, it has detected that the computation of M on input $w = xbav$ begins with a sequence of two not strictly monotone cycles that delete the factor ba. In this case A sets CRS to q_{i_2}, it replaces its current state table by the table T that was stored in B together with the pair

$(b, (q_{i_1}, p_{i_1}, q_{i_2}))$, and it empties the buffer B. Thus, A now simulates the computation of M that begins with the restarting configuration $q_{i_2}\mathbin{¢}xv\$$, and as M is deterministic, A can do so by starting with the first letter of v and by considering the states of M that are reached by moving right across the prefix $¢x$, which are already recorded in the table T.

We now give a formal description of the DFA $A = (Q_A, \Sigma \cup \{\$\}, \delta_A, q_0^{(A)}, F)$, where we take the fact into account that sequences of not strictly monotone cycles within a computation of M can be of any length up to n.

Let $Q_r \subseteq Q$ be the set of *restart states* of M, that is, $q \in Q$ belongs to Q_r, if $q = q_0$, or if $(q, \varepsilon) \in \delta(p, a)$ for some $p \in Q$ and $a \in \Sigma$. Further, let $k \in \mathbb{N}$ such that $|Q_r| = k + 1$. W.l.o.g. we may assume that $Q_r = \{q_0, q_1, \ldots, q_k\}$. Note that the states in Q_r can be seen as the initial states of $k + 1$ 'finite state automata'. Now the *set of states* Q_A of the DFA A is defined as

$$Q_A = \{[q, T, B] \mid q \in Q_r\},$$

where $T \subseteq \{(q_i, q_i') \mid q_i \in Q_r \text{ and } q_i' \in Q \cup \{-\}\}$ is a *state table* that mirrors a parallel computation of these different finite-state acceptors on a prefix of the current input of M, and B is the buffer mentioned above which is realized as a $(k + 1) \times (k + 2)$ matrix. The rows of B are indexed by the restart states $q_\mu \in Q_r$, that is, for each $\mu \in \{0, \ldots, k\}$, row μ corresponds to the restart state q_μ. For all μ, row μ will, at each moment in time, either contain the marker $-$ in all entries, or it will contain a state table T_μ in its first entry, a sequence of tuples

$$(b_0, (q_{i_0}, q_{i_0}', q_{i_0}''))(b_1, (q_{i_1}, q_{i_1}', q_{i_0})) \ldots (b_{s-1}, (q_{i_{s-1}}, q_{i_{s-1}}', q_{i_{s-2}}))(b_s, (q_{i_s}, q_{i_s}', q_{i_{s-1}}))$$

in columns 2 to $s+2$ such that $s \geq 0$, $b_0, \ldots, b_s \in \Sigma$, $q_{i_0}, \ldots, q_{i_s}, q_{i_0}'' \in Q_r$, and $q_{i_0}', \ldots, q_{i_s}' \in Q$, where $q_{i_s} = q_\mu$, possibly some entries of the form $(c_1) \ldots (c_{s'})$, $c_1, \ldots, c_{s'} \in \Sigma$, in columns $s+3$ to $s+s'+3$, and the marker $-$ in all columns $j > s+s'+3$. Here the above sequence of tuples describes a possible sequence of not strictly monotone cycles that, starting in restart state q_μ, would delete the factor $b_0 \ldots b_s$, which is followed by the word $c_1 \ldots c_{s'}$. Observe that the restart state q_μ that corresponds to row μ occurs as the first state in the last tuple $(b_s, (q_{i_s}, q_{i_s}', q_{i_{s-1}}))$, and that the third state in a tuple is just the restart state that occurred as the first state in the previous tuple. This mirrors the fact that, in the aforementioned sequence of not strictly monotone cycles, the factor $b_0 \ldots b_s$ would be deleted letter by letter from right to left.

Initially A is in state

$$q_0^{(A)} = \left[q_0, \{(q_0, \langle\delta(q_0, ¢)\rangle), \ldots, (q_k, \langle\delta(q_k, ¢)\rangle), \begin{pmatrix} - & \cdots & - \\ \vdots & \ddots & \vdots \\ - & \cdots & - \end{pmatrix} \right],$$

where $\langle\delta(q_i, ¢)\rangle$ $(0 \leq i \leq k)$ is the state that is reached by M from state q_i on seeing the ¢-symbol, that is, $\langle\delta(q_i, ¢)\rangle = q_i'$, if $\delta(q_i, ¢) = (q_i', \mathsf{MVR})$, and it is undefined (denoted by $-$), if $\delta(q_i, ¢)$ is undefined, and all entries of the buffer are filled with the marker $-$.

The description of the *transition function* δ_A of A can be divided into three parts, where the first part concerns the case that on the current input letter only move-right steps and undefined transitions of M occur, the second part deals with the case that on this particular input letter some rewrite/restart operations are possible, but none of them concerns the currently active restart state, and the third part deals with the case that the currently active restart state leads to a rewrite/restart step on the current input letter. Let us assume that A has already processed a prefix x of the given input, and now it is to process the next letter a. While reading x, A has reached a state $[q_r, T, B]$ from its initial state $q_0^{(A)}$, where $q_r \in Q_r$ is the restart state in which the cycle of M started that is currently being simulated, $T = \{(q_0, q_0'), \ldots, (q_k, q_k')\}$ is the current state table, where, for all $i = 0, \ldots, k$, q_i' is the state that M would reach starting from the configuration $q_i ¢ x$ by executing $|x|+1$ MVR-steps, if that is possible, and otherwise $q_i' = -$, and B is the current buffer that may contain information on previous rewrite/restart steps that correspond to possible sequences of not strictly monotone cycles that would delete a factor of x.

(a) If, for all $i \in \{0, \ldots, k\}$, $\delta(q_i', a)$ is either a MVR-step or it is undefined, then

$$\delta_A([q_r, T, B], a) = \left[q_r, T', \begin{pmatrix} - & \cdots & - \\ \vdots & \ddots & \vdots \\ - & \cdots & - \end{pmatrix} \right], \tag{2.1}$$

where $T' = \{(q_0, \langle\delta(q_0', a)\rangle), ..., (q_k, \langle\delta(q_k', a)\rangle)\}$, and the buffer B is completely emptied. Here $\langle\delta(q_i', a)\rangle$ is the state q_i'' that is reached by M from state q_i' on seeing the symbol a, if $\delta(q_i', a) = (q_i'', \mathsf{MVR})$, and it is undefined (denoted by $-$), if $\delta(q_i', a)$ is undefined. Thus, for all $i = 0, \ldots, k$, $\langle\delta(q_i', a)\rangle$ is the state that M would reach starting from the configuration $q_i ¢ x a$ by executing $|x| + 2$ MVR-steps, if that is possible, and otherwise it is $-$.

(b) If, for one or more $i \in \{0, \ldots, k\}$, $\delta(q'_i, a)$ is a rewrite/restart step, then the situation gets much more complicated. Here we consider the case that $\delta(q'_{i_j}, a) = (q''_{i_j}, \varepsilon)$ for some $j = 1, \ldots, s$, that $\delta(q'_i, a)$ is a MVR-step or it is undefined for all other values of i, and that $q_r \notin \{q_{i_1}, \ldots, q_{i_s}\}$, that is, none of the above rewrite/restart steps corresponds to the current restart state. Assume that

$$B = \begin{pmatrix} T_0: & l_{0,0} & l_{0,1} & \ldots & l_{0,k} \\ T_1: & l_{1,0} & l_{1,1} & \ldots & l_{1,k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ T_k: & l_{k,0} & l_{k,1} & \ldots & l_{k,k} \end{pmatrix},$$

where, for each $\mu \in \{0, \ldots, k\}$, row μ of B is either filled with a state table T_μ in the first column, tuples of the form $(b, (q, p, q'))$ in all columns up to some column $v \geq 2$, possibly some entries of the form (c) in all columns $v < \rho \leq v + v'$, and the marker $-$ in all entries $l_{\mu,\rho}$ with $\rho > v + v'$, or all entries of row μ contain the marker $-$. Notice again that the first column differs from the others, as it contains state tables. However, if there are no tuples in a particular row, then also this entry is set to $-$. Now δ_A is defined by taking

$$\delta_A([q_r, T, B], a) = [q_r, T', B'], \tag{2.2}$$

where $T' = \{(q_0, q''_0), \ldots, (q_k, q''_k)\}$ is obtained from $T = \{(q_0, q'_0), \ldots, (q_k, q'_k)\}$ by replacing, for all $i = 0, \ldots, k$, q'_i by q''_i as follows:

$$q''_i = \begin{cases} -, & \text{if } q'_i = -, \\ -, & \text{if } \delta(q'_i, a) \text{ is a rewrite/restart step,} \\ q, & \text{if } \delta(q'_i, a) = (q, \mathsf{MVR}), \\ -, & \text{if } \delta(q'_i, a) \text{ is undefined.} \end{cases}$$

Then, for all $i = 0, \ldots, k$, q''_i is the state that M would reach starting from the configuration $q_i \mathbin{¢} xa$ by executing $|x| + 2$ MVR-steps, if that is possible, and otherwise $q''_i = -$.

The buffer B contains information on rewrite steps that correspond to possible sequences of not strictly monotone cycles of M that would delete a factor of x. The new buffer B' is obtained from B by combining the information on the rewrite/restart steps $\delta(q'_{i_j}, a) = (q''_{i_j}, \varepsilon)$ $(1 \leq j \leq s)$ with the information stored in B.

For each $j = 1, \ldots, s$, if the row of B that corresponds to the restart state q_{i_j} is not empty (that is, it does not only contain entries of the form $-$), then all entries of the corresponding row of B' are set to $-$, as in this case we have a repetition of restart states. Otherwise, this row of B' is defined as follows. If the row of B that corresponds to the restart state q''_{i_j} contains a sequence of entries

$$T_\mu : (b_0, (p_0, p'_0, p''_0)) \ldots (b_t, (q''_{i_j}, p'_t, p''_t)),$$

then the row of B' that corresponds to the restart state q_{i_j} is set to

$$T_\mu : (b_0, (p_0, p'_0, p''_0)) \ldots (b_t, (q''_{i_j}, p'_t, p''_t))(a, (q_{i_j}, q'_{i_j}, q''_{i_j})). \tag{2.3}$$

Observe how the last two tuples match in the way that the former tuple describes a cycle of M that starts in the restart state q''_{i_j}, and the new tuple describes a cycle of M that ends with a rewrite/restart step that leads to this very restart state.

If the row of B that corresponds to the restart state q''_{i_j} is empty, then the row of B' that corresponds to the restart state q_{i_j} is set to

$$T : (a, (q_{i_j}, q'_{i_j}, q''_{i_j})). \tag{2.4}$$

In each of these two cases, this row of B' describes a possible sequence of not strictly monotone cycles of M that would delete a suffix of xa. Further, if the row of B that corresponds to the restart state q''_{i_j} contains a sequence of entries

$$T_\mu : (b_0, (p_0, p'_0, p''_0)) \ldots (b_t, (q''_{i_j}, p'_t, p''_t))(b_{t+1}) \ldots (b_{t+t'})$$

for some $t' \geq 1$, then all entries in the row of B' that correspond to the restart state q_{i_j} are set to $-$, as in this case the cycle ending with the rewrite/restart step $\delta(q'_{i_j}, a) = (q''_{i_j}, \varepsilon)$ would be followed by a sequence of cycles that would delete the prefix $b_0 \ldots b_t$ of the suffix $b_0 \ldots b_t b_{t+1} \ldots b_{t+t'}$ of x, that is, we would have a non-monotone computation.

Finally, for all μ such that row μ does not correspond to any of the restart states q_{i_j} $(1 \leq j \leq s)$, if row μ of B contains a sequence of entries

$$T_\mu : (c_0, (p_0, p'_0, p''_0)) \ldots (c_t, (q_t, p'_t, p''_t))(c_{t+1}) \ldots (c_{t+t'})$$

for some $t, t' \geq 0$, then row μ of B' is set to

$$T_\mu : (c_0, (p_0, p_0', p_0'')) \dots (c_t, (q_t, p_t', p_t''))(c_{t+1}) \dots (c_{t+t'})(a), \tag{2.5}$$

in all other cases all entries in row μ of B' are set to $-$. In the former case this row describes the fact that there exists a possible sequence of not strictly monotone cycles that would delete the prefix $c_0 \dots c_t$ of the suffix $c_0 \dots c_t c_{t+1} \dots c_{t+t'} a$ of xa.

(c) Finally we consider the case that $\delta(q_{i_j}', a) = (q_{i_j}'', \varepsilon)$ for some $j = 1, \dots, s$, that $\delta(q_i', a)$ is a MVR-step or it is undefined for all other values of i, and that $q_r = q_{i_j}$ for some $j \in \{1, \dots, s\}$, that is, one of the s computations that perform the rewrite/restart step on the current letter a corresponds to the actual computation that is currently being simulated. In this case the current rewrite/restart step completes a cycle which may be part of a not strictly monotone sequence of cycles that delete a suffix of xa. To simplify the notation we assume that $q_r = q_{i_1}$, that is, $\delta(q_{i_1}', a) = (q_{i_1}'', \varepsilon)$ is the rewrite/restart step that M would execute in the current cycle. Further, let

$$B = \begin{pmatrix} T_0 : & l_{0,0} & l_{0,1} & \dots & l_{0,k} \\ T_1 : & l_{1,0} & l_{1,1} & \dots & l_{1,k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ T_k : & l_{k,0} & l_{k,1} & \dots & l_{k,k} \end{pmatrix},$$

where, for each $\mu \in \{0, \dots, k\}$, row μ of B is either filled with a state table T_μ in the first column, tuples of the form $(b, (q, p, q'))$ in all columns up to some column $v \geq 2$, possibly some entries of the form (c) in all columns $v < \rho \leq v + v'$, and the marker $-$ in all entries $l_{\mu,\rho}$ with $\rho > v + v'$, or all entries of row μ contain the marker $-$. Now δ_A is defined by taking

$$\delta_A([q_r, T, B], a) = [q_r', T', B'], \tag{2.6}$$

where q_r', T', and B' are described below.

First we consider the definition of the restart state q_r' and the state table T'. If the row of B that corresponds to the restart state q_{i_1}'' contains a sequence of entries

$$T_\mu : (b_0, (p_0, p_0', p_0'')) \dots (b_t, (q_{i_1}'', p_t', p_t'')),$$

then A has discovered a sequence of not strictly monotone cycles of M that delete the

suffix $b_0 \dots b_t a$ of xa. Accordingly, $q'_r = p''_0$ and $T' = T_\mu$ are chosen, as T_μ describes the behavior of M concerning MVR-steps that read across the prefix $\not\vdash x_1$ of $\not\vdash xa$ that is obtained by deleting the suffix $b_0 \dots b_t a$. In all other cases, $q'_r = q''_{i_1}$ and $T' = T$ are chosen, as in this case the next cycle, which begins with the restart state q''_{i_1}, does not belong to a sequence of not strictly monotone cycles that would delete a suffix of x.

The matrix B' is defined as follows. If the row of B that corresponds to the restart state q''_{i_1} contains a sequence of entries

$$T_\mu : (b_0, (p_0, p'_0, p''_0)) \dots (b_t, (q''_{i_1}, p'_t, p''_t)), \tag{2.7}$$

then all entries in the row of B' that correspond to the restart state q''_{i_1} are set to the marker $-$. For any other row μ' of B, we proceed as follows:

- if row μ' of B contains a sequence of the form

  $$T_{\mu'} : (b_i, (p_i, p'_i, p''_i)) \dots (b_s, (p_s, p'_s, p''_s))(b_{s+1}) \dots (b_t),$$

  for some $i \geq 0$ and some $i \leq s \leq t$, then all entries in the corresponding row of B' are set to $-$;
- if row μ' of B contains a sequence of the form

  $$T_{\mu'} : (c_0, (p_0, p'_0, p''_0)) \dots (c_s, (p_s, p'_s, p''_s))(c_{s+1}) \dots (c_{s+s'})(b_0) \dots (b_t)$$

  for some $s \geq 0$ and $s' \geq 1$, then the corresponding row of B' is set to

  $$T_{\mu'} : (c_0, (p_0, p'_0, p''_0)) \dots (c_s, (p_s, p'_s, p''_s))(c_{s+1}) \dots (c_{s+s'});$$

- if row μ' of B contains a sequence of the form

  $$T_{\mu'} : (c_0, (p_0, p'_0, p''_0)) \dots (c_s, (p_s, p'_s, p''_s))(b_0, (\dots)) \dots (b_t, (\dots))$$

  for some $s \geq 0$, where the subsequence $(b_0, (\dots)) \dots (b_t, (\dots))$ consists of tuples only, or it consists of $r \geq 0$ tuples that are followed by the sequence $(b_r)(b_{r+1}) \dots (b_t)$, then the row of B' that corresponds to the restart state p_s is set to

  $$T_{\mu'} : (c_0, (p_0, p'_0, p''_0)) \dots (c_s, (p_s, p'_s, p''_s));$$

– finally, all other rows of B' will only contain the marker $-$.

If, however, the row of B that corresponds to the restart state q_{i_1} does not have the form as in equation (2.7), then we take $B' = B$.

Finally, A enters its accepting state, if the current cycle leads M to accept on seeing the \$-symbol, that is, $\delta_A([q_r, T, B], \$) = [\mathsf{Accept}]$ if and only if $(q_r, q) \in T$ for some state such that $\delta(q, \$) = \mathsf{Accept}$.

In summary the overall computation of A on input w proceeds as follows. Assume that A reads the letter a.

- If, for no pair (q_i, q_i') stored in the table T, $\delta(q_i', a)$ is a rewrite operation, then the table T is updated by replacing (q_i, q_i'), for all i, by $(q_i, \hat{q}_i)$, if $\delta(q_i', a) = (\hat{q}_i, \mathsf{MVR})$, and by $(q_i, -)$, if $\delta(q_i', a)$ is undefined. In this situation, CRS remains unchanged, and the buffer is emptied completely (see (a)).
- If CRS is q_i, and for the pair (q_i, q_i') stored in T, $\delta(q_i', a) = (q_j, \varepsilon)$, then the rewrite operation of the current cycle of M has been detected. If B is empty, then CRS is set to q_j, and T and B remain unchanged, and if B is non-empty, then A proceeds as detailed in (c).
- If $\delta(q_i', a) = (q_j, \varepsilon)$ for a pair (q_i, q_i') stored in T, where q_i is different from the state stored in CRS, then the corresponding information is pushed onto the buffer B as detailed in (b).

The finite-state acceptor A keeps on reading the word w letter by letter from left to right until it encounters the right sentinel \$. Now the actual state of A is accepting, if the pair (q_i, q_i') in the current table T that corresponds to the current state q_i of M stored in CRS satisfies the condition that $\delta(q_i', \$) = \mathsf{Accept}$.

Notice again that the space provided for the buffer suffices for the task described above. Of course, it may happen that a row of B is completely filled with tuples. As a not strictly monotone sequence of cycles within any computation of M can consist of only up to $k+1$ cycles, it follows that the sequence corresponding to a completely filled row of B cannot be reached within any computation of M. Hence, we can adjust our construction in that we empty any row of B as soon as it contains two tuples with the same restart state.

As stated above our initial example mirrors only a small part of the possible computations of M. In general the situation can be much more complicated. In fact, there are three major cases that we need to deal with.

Case 1. The word w has a factorization of the form $w = xa_m a_{m-1} \cdots a_1 y$ such that the computation of M on input w begins with a sequence of m cycles such that, in the j-th cycle ($1 \leq j \leq m$), the letter a_j is deleted. Thus, while the first cycle has the form

$$q_0 \text{¢} w\$ = q_0 \text{¢} x a_m \cdots a_2 a_1 y\$ \vdash^+_{\mathsf{MVR}} \text{¢} x a_m \cdots a_2 p_0 a_1 y\$ \vdash q_{i_1} \text{¢} x a_m \cdots a_2 y\$,$$

the j-th cycle ($2 \leq j \leq m$) has the form

$$q_{i_{j-1}} \text{¢} x a_m \cdots a_{j+1} a_j y\$ \vdash^+_{\mathsf{MVR}} \text{¢} x a_m \cdots a_{j+1} p_{i_{j-1}} a_j y\$ \vdash q_{i_j} \text{¢} x a_m \cdots a_{j+1} y\$.$$

This case is treated in analogy to the example above.

Case 2. The word w has a factorization of the form $w = xa_m \cdots a_{s+1} a_s y$ such that, for each $j = s+1, \ldots, m$, there is a possible cycle of M of the form

$$q_{i_{j-1}} \text{¢} x a_m \cdots a_{j+1} a_j a_s y\$ \vdash^+_{\mathsf{MVR}} \text{¢} x a_m \cdots a_{j+1} p_{i_{j-1}} a_j a_s y\$ \vdash q_{i_j} \text{¢} x a_m \cdots a_{j+1} a_s y\$,$$

but from the restarting configuration $q_i \text{¢} x a_m \cdots a_{s+1} a_s y\$$, M does not apply a rewrite operation to the letter a_s for any state $q_i \in Q_r$. While reading the prefix $xa_m \cdots a_{s+1}$, A has collected the following information in its finite-state control:

- CRS is q_0;
- the table T contains the pairs of the form (q_i, q_i') $(i = 0, 1, \ldots, k)$, where q_i' is the state that M reaches from the restarting configuration $q_i \text{¢} w\$ = q_i \text{¢} x a_m \cdots a_s v\$$ by moving right across the prefix $\text{¢} x$;
- the buffer B contains the following sequence in the row i_s that corresponds to restart state q_{i_s}:

$$T_{i_s} : (a_m, (q_{i_{m-1}}, p_{i_{m-1}}, q_{i_m}))\,(a_{m-1}, (q_{i_{m-2}}, p_{i_{m-2}}, q_{i_{m-1}})) \cdots (a_{s+1}, (q_{i_s}, p_{i_s}, q_{i_{s+1}})).$$

On encountering the letter a_s, A realizes that M cannot execute a rewrite step that completes the partial computation on input w consisting of the sequence of cycles encoded by the i_s-th row of B given above in the sense of Case 1. In fact, as M cannot execute any

rewrite step on a_s, no matter in which state it starts the current cycle, monotonicity of M implies that the partial computation encoded by the i_s-th row of B cannot be a part of the computation of M on input w. Therefore, A empties the buffer B completely (see (a)), and T is updated by replacing each pair (q_i, q'_i) in T by the pair $(q_i, \hat{q}_i)$, if starting from the restarting configuration $q_i\mathrm{¢}xa_m \cdots a_{s+1}a_s v\$$, M reaches state $\hat{q}_i$ by moving right across the prefix $\mathrm{¢}xa_m \cdots a_s$ (see equation (2.1)).

Case 3. This case is a combination of the two cases above. The word w has a factorization of the form $w = xa_m \cdots a_{s+1}a_s a_{s-1} \cdots a_1 y$ such that all of the following properties hold:

(α) For each $j = s+1, \ldots, m$, there is a possible cycle of M of the form

$$q_{i_{j-1}}\mathrm{¢}x \cdots a_j a_s \cdots a_1 y\$ \vdash^+_{\mathsf{MVR}} \mathrm{¢}x \cdots p_{i_{j-1}} a_j a_s \cdots a_1 y\$ \vdash q_{i_j}\mathrm{¢}x \cdots a_{j+1}a_s \cdots a_1 y\$.$$

(β) There is a possible cycle of the form

$$\begin{aligned} &q'_{i_{s-1}}\mathrm{¢}xa_m \cdots a_{s+1}a_s \cdots a_1 y\$ \vdash^+_{\mathsf{MVR}} \mathrm{¢}xa_m \cdots a_{s+1}p'_{i_{s-1}}a_s \cdots a_1 y\$ \\ &\vdash q'_{i_s}\mathrm{¢}xa_m \cdots a_{s+1} \cdots a_1 y\$, \end{aligned}$$

but for each such cycle, $q'_{i_s} \neq q_{i_s}$.

(γ) For each $\mu = 1, \ldots, s-1$, there is a possible cycle of M of the form

$$q'_{i_{\mu-1}}\mathrm{¢}xa_m \cdots a_{\mu+1}a_\mu y\$ \vdash^+_{\mathsf{MVR}} \mathrm{¢}xa_m \cdots a_{\mu+1}p'_{i_{\mu-1}}a_\mu y\$ \vdash q'_{i_\mu}\mathrm{¢}xa_m \cdots a_{\mu+1}y\$$$

such that q'_{i_0} coincides with the current value of CRS.

Thus, there is a possible sequence of not strictly monotone cycles that would delete the factor $a_m \cdots a_{s+1}$ of w by (α), there is a cycle that would delete the letter a_s by (β), but the restart state q'_{i_s} reached by the latter cycle does *not* coincide with the initial state q_{i_s} of the first of the former sequence of cycles, and there is a sequence of not strictly monotone cycles that deletes the factor $a_{s-1} \cdots a_1$ of w by (γ) such that the initial state q'_{i_0} of the first of these cycles coincides with the restart state of the current cycle of M that is being simulated.

While reading the prefix $xa_m \cdots a_{s+1}$, A has collected the same information as in Case 2. On encountering the letter a_s, A realizes that M cannot execute another cycle that would extend the partial computation on input w consisting of the sequence of cycles of (α). As,

however, there is a rewrite operation that M may apply to the letter a_s, A continues as follows (see (b)):

- CRS remains unchanged;
- the table T is updated as above (see equation (2.2));
- all rows of the buffer B that contain any entries other than $-$ are extended by the entry (a_s) (see equation (2.5)), and each row that corresponds to a restart state $q'_{i_{s-1}}$ is initialized with the entries T and $(a_s, (q'_{i_{s-1}}, p'_{i_{s-1}}, q'_{i_s}))$ (see equation (2.4)).

For each letter $a_{s-1}, \ldots, a_2$, A extends the buffer B as described in (b). Observe that m, the length of the factor $a_m \ldots a_{s+1} a_s \ldots a_1$, must still satisfy the condition $m \leq k+1$ due to the fact that M is deterministic. On encountering the letter a_1, A realizes that it has found a sequence of s cycles that is the first part of the computation of M on input w. Accordingly, it acts similar to Case 1. However, it must delete from B all those entries that correspond to the cycles deleting the factor $a_s \ldots a_1$ (see (c)). Of course, the above situation may occur repeatedly, but the overall length of the longest sequence of possible rewrite operations that is stored in B is always bounded in length from above by the number $k+1$ of restart states of M. This completes the description of Case 3.

It follows that $L(A) = L \cdot \$$. Since the class of regular languages is closed under right quotients, this implies that the language L is regular, too. This completes the proof of Theorem 2.2.11. □

Next we extend Theorem 2.2.11 to non-deterministic nf-R-automata.

Theorem 2.2.12. $\mathcal{L}(\text{mon-nf-R}(1)) = \text{REG}$.

Proof. Let $M = (Q, \Sigma, ¢, \$, q_0, 1, \delta)$ be a mon-nf-R(1)-automaton for $L \subseteq \Sigma^*$. Then M can be simulated by a non-deterministic finite state automata (NFA) $A = (Q_A, \Sigma \cup \{\$\}, \delta_A, q_0^{(A)}, F)$ for the language $L \cdot \$$ by using exactly the same strategy as in the proof of Theorem 2.2.11. In each step the NFA A just has to guess the transition step that M is going to execute. However, as in the proof of Theorem 2.2.11 determinism was used as an essential argument to obtain the upper bound for the size of the buffer B implemented as part of the finite-state control of the DFA constructed, we must still verify the following claim for the non-deterministic case.

Claim. Let $w = xa_m a_{m-1} \cdots a_1 y$ be an input word such that M has a computation on input w that begins with a sequence of m cycles such that, in the j-th cycle ($1 \leq j \leq m$), the letter a_j is deleted. Then $m \leq k+1$, where $k+1$ is the number of restart states of M (see above).

Proof. Consider the computation of M on input w for which the first cycle has the form

$$q_0 \cent w \$ = q_0 \cent x a_m \cdots a_2 a_1 v \$ \vdash^+_{\mathsf{MVR}} \cent x a_m \cdots a_2 p_0 a_1 y \$ \vdash q_{i_1} \cent x a_m \cdots a_2 y \$,$$

and the j-th cycle ($2 \leq j \leq m$) has the form

$$q_{i_{j-1}} \cent x a_m \cdots a_{j+1} a_j y \$ \vdash^+_{\mathsf{MVR}} \cent x a_m \cdots a_{j+1} p_{i_{j-1}} a_j y \$ \vdash q_{i_j} \cent x a_m \cdots a_{j+1} y \$.$$

Assume that $m > k+1$ holds. Then there exist two indices $0 \leq \mu < \nu \leq m$ such that the restart states q_{i_μ} and q_{i_ν} are identical. Here we simply denote the initial state q_0 as q_{i_0}. Hence, M can also perform the following computation:

$$\begin{array}{lll}
q_0 \cent x a_m \cdots a_2 a_1 y \$ & \vdash_M^{c^{\mu-1}} & q_{i_{\mu-1}} \cent x a_m \cdots a_{\nu+2} a_{\nu+1} a_\nu \cdots a_{\mu+1} a_\mu y \$ \\
& \vdash^+_{\mathsf{MVR}} & \cent x a_m \cdots a_{\nu+2} a_{\nu+1} a_\nu \cdots a_{\mu+1} p_{i_{\mu-1}} a_\mu y \$ \\
& \vdash_M & q_{i_\mu} \cent x a_m \cdots a_{\nu+2} a_{\nu+1} a_\nu \cdots a_{\mu+1} y \$ \\
& = & q_{i_\nu} \cent x a_m \cdots a_{\nu+2} a_{\nu+1} a_\nu \cdots a_{\mu+1} y \$ \\
& \vdash^+_{\mathsf{MVR}} & \cent x a_m \cdots a_{\nu+2} p_{i_\nu} a_{\nu+1} a_\nu \cdots a_{\mu+1} y \$ \\
& \vdash_M & q_{i_{\nu+1}} \cent x a_m \cdots a_{\nu+2} a_\nu \cdots a_{\mu+1} y \$.
\end{array}$$

The cycle starting with the configuration $q_{i_{\mu-1}} \cent x a_m \cdots a_{\nu+2} a_{\nu+1} a_\nu \cdots a_{\mu+1} a_\mu y \$$ has right distance $d_1 = |y| + 2$, while the next cycle, that is, the one starting with the configuration $q_{i_\nu} \cent x a_m \cdots a_{\nu+2} a_{\nu+1} a_\nu \cdots a_{\mu+1} y \$$, has right distance $d_2 = |y| + 2 + \nu - \mu > d_1$. This, however, contradicts our assumption that M is monotone, which means that each and every computation of M that starts from an initial configuration is monotone. Thus, it follows that $m \leq k+1$ holds as in the deterministic case considered in Theorem 2.2.11. This completes the proof of the claim above.

Hence, it follows that, whenever in a simulation of a computation of M by A a sequence of cycles is detected that contains a repetition of a restart state, then this sequence of cycles cannot possibly be a part of a computation of M that begins with a proper initial configuration. Accordingly, the actual computation of A can be terminated in a non-accepting state. It follows that $L(A) = L \cdot \$$, which in turn implies that the language L itself is regular. □

Finally we want to extend Theorem 2.2.11 even to non-forgetting RR-automata. For doing so we need the following technical result on deterministic two-way finite state automata from [AHU69] (see pages 212–213). Here note that a deterministic *two-way finite state automaton* (2DFA for short) $A = (Q, \Sigma, \delta, q_0, F)$ is simply a DFA, where the transition function is extended to $\delta : Q \times \Sigma \to Q \times \{L, R\}$ (e.g. exposed in [HU79]).

Lemma 2.2.13. *Let B be a* DFA*. For each word x and each integer i, $1 \leq i \leq |x|$, let $q_B(x, i)$ be the internal state of B after processing the prefix of length i of x. Then there exists a* 2DFA *B' such that, for each input x and each $i \in \{2, 3, \ldots, |x|\}$, if B' starts its computation on x in a state corresponding to $q_B(x, i)$ with its head on the i-th symbol of x, then B' finishes its computation in a state that corresponds to $q_B(x, i-1)$ with its head on the $(i-1)$-th symbol of x. During this computation B' only visits (a part of) the prefix of length i of x.*

Informally, a 2DFA is able to recalculate the current state and the position of the head of a corresponding DFA, after making an excursion to the left (end) of the tape.

Theorem 2.2.14. $\mathcal{L}(\text{det-mon-nf-RR}(1)) = \text{REG}$.

Proof. Obviously it remains to prove the inclusion from left to right. So let $M = (Q, \Sigma, ¢, \$, q_0, 1, \delta)$ be a det-mon-nf-RR(1)-automaton for $L \subseteq \Sigma^*$. W.l.o.g. we may assume that $Q = \{q_0, q_1, \ldots, q_{n-1}\}$, and that M executes restart- and accept-instructions only on reading the \$-symbol. Below we describe a 2DFA $A = (Q_A, \Sigma \cup \{¢, \$\}, \delta_A, q_0^{(A)}, F)$ for $¢ \cdot L \cdot \$$. Essentially, A works in the very same way as the DFA in the proof of Theorem 2.2.11, but there is a technical problem that we must solve:

Whenever M executes a rewrite operation, then we need to know whether this operation is within an accepting or a rejecting tail computation, or whether it is contained in a cycle of M. In the latter case, we also need to know which state of M is entered by the restart operation of this cycle.

To solve this problem A will execute a preprocessing stage given an input of the form $¢w\$$.

Preprocessing Stage: Let $w \in \Sigma^*$, and let $¢w\$$ be the input for A.

Step 1. Starting from the initial configuration $q_0^{(A)}¢w\$$, A scans its input from left to right until it encounters the \$-symbol, that is, $q_0^{(A)}¢w\$ \vdash_A^+ ¢wq_1^{(A)}\$$ for some state $q_1^{(A)} \in Q_A$.

Step 2. For each suffix v of w, and for each state $q \in Q$, let

$$P_{v\$}(q) = \{p \in Q \mid \exists p' \in Q : ¢pv\$ \vdash_{\text{MVR}}^{|v|} ¢vp'\$ \vdash_{\text{Restart}} q¢v\$\},$$

and let

$$P_{v\$}(+) = \{p \in Q \mid \exists p' \in Q : ¢pv\$ \vdash_{\mathsf{MVR}}^{|v|} ¢vp'\$ \vdash \mathsf{Accept}\}.$$

The initial sets

$$P_{\$}(q) = \{p \in Q \mid \delta(p, \$) = (q, \mathsf{Restart})\} \text{ and } P_{\$}(+) = \{p \in Q \mid \delta(p, \$) = \mathsf{Accept}\},$$

which are easily obtained from M, are stored in A's finite-state control.

Step 3. A reads its tape from right to left, letter by letter. Assume that $w = xav$, where $x, v \in \Sigma^*$ and $a \in \Sigma$, and that A has already moved left across the suffix v thereby computing the sets $P_{v\$}(q)$ for all $q \in Q$ and $P_{v\$}(+)$. Now A moves left reading the symbol a, and while doing so it updates the set $P_{v\$}(q)$ to

$$P_{av\$}(q) = \{p \in Q \mid \exists p' \in P_{v\$}(q) : \delta(p, a) = (p', \mathsf{MVR})\}$$

for each $q \in Q$, and it updates the set $P_{v\$}(+)$ to

$$P_{av\$}(+) = \{p \in Q \mid \exists p' \in P_{v\$}(+) : \delta(p, a) = (p', \mathsf{MVR})\,\}.$$

This process continues until A reaches the ¢-symbol. At that moment it has stored the sets $P_{w\$}(q)$ $(q \in Q)$ and $P_{w\$}(+)$ in its finite-state control.

Unfortunately, A can only store one collection (or rather, a finite number of collections) of these sets in its finite state control, that is, when storing the sets $P_{av\$}(q)$ $(q \in Q)$ and $P_{av\$}(+)$, it forgets the sets $P_{v\$}(q)$ $(q \in Q)$ and $P_{v\$}(+)$. Fortunately, we can now apply Lemma 2.2.13 to the DFA realizing Steps 2 and 3 of the above preprocessing stage. Just observe that this automaton works from right to left, and so we need the symmetric version of the above lemma. This means that from the sets $P_{av\$}(q)$ $(q \in Q)$ and $P_{av\$}(+)$, the sets $P_{v\$}(q)$ $(q \in Q)$ and $P_{v\$}(+)$ can be recomputed.

Finally, we combine the 2DFA of the preprocessing stage above with the DFA from the proof of Theorem 2.2.11. For each rewrite operation of the form $\delta(q', a) = (p, \varepsilon)$ encountered in the simulation of M, we check whether $p \in P_{v\$}(+)$ or whether $p \in P_{v\$}(q)$ for some $q \in Q$, where v is the corresponding suffix of the input word w. Based on this information the simulation then continues as in the proof of Theorem 2.2.11. □

In summary we identified three new types of restarting automata that characterize the regular languages. Further, note again that already non-deterministic RR(1)-automata

compute non-regular languages. Hence, all other variants of restarting automata with window size one considered in Subsection 2.2 lead to super-classes of the regular languages.

We close this subsection and therewith also the section by shortly addressing the question why we are interested in new characterizations for regular languages in terms of restarting automata. Without anticipating results given in Chapter 4, restarting automata of those types above form a very fruitful basis for computing relations, both in a theoretical and in a practical sense. One reason for that is the succinctness of their representations of regular languages. We know from Kutrib and Reimann ([KR08, Rei07]) that in terms of descriptional complexity, there is a benefit in using (forgetting) restarting automata to represent regular languages. In particular, there are exponential trade offs for changing from R(1)- or det-RR(1)-automata to NFA. Here, as a first preliminary result we can show that non-forgetting restarting automata yield even more succinct representations than (forgetting) restarting automata.

Proposition 2.2.15. *For each $n \geq 2$, there exists a language $L_n \subseteq \{a,b\}^*$ that is accepted by a* det-mon-nf-RR(1)*-automaton with $O(n)$ states, but every* det-RR(1)*-automaton accepting L_n has $\Omega(2^n)$ many states.*

Proof. For $n \geq 2$, let $L_n = \{w \in \{a,b\}^m \mid m > n,\ w_n = a,\ \text{and}\ w_{m+1-n} = b\}$, where w_i $(1 \leq i \leq |w|)$ denotes the i-th symbol of w. A det-mon-nf-RR(1)-automaton M for L_n can be described by the following meta-instructions, where $x \in \{a,b\}$:

$$\begin{array}{llllll} (1) & (q_0, & \mathrm{¢}, & x \to \varepsilon, & \{a,b\}^{n-2} \cdot a \cdot \{a,b\}^+ \cdot \$, & q_1), \\ (2) & (q_1, & \mathrm{¢} \cdot a^*, & b \to \varepsilon, & \{a,b\}^{n-1} \cdot \{a,b\}^+ \cdot \$, & q_1), \\ (3) & (q_1, & \mathrm{¢} \cdot a^*, & b \to \varepsilon, & \{a,b\}^{n-1} \cdot \$, & \mathsf{Accept}). \end{array}$$

For realizing these meta-instructions $O(n)$ states suffice, as M must be able to count from 1 to n.

Now let $M'_n = (Q', \{a,b\}, \mathrm{¢}, \$, q'_0, 1, \delta')$ be a det-RR(1)-automaton such that $L(M') = L_n$. We consider the accepting computation of M'_n given an input of the form $w = xabyabz$ such that $|x| + |y| + 2 = n - 1 = |y| + |z| + 2$. Then $w \in L_n$, and hence, the computation of M'_n on input w is accepting. If this computation begins with a cycle, then it has the form

$$q_0\mathrm{¢}\,w\$ = q_0\mathrm{¢}\,xabyabz \vdash^c_{M'_n} q_0\mathrm{¢}\,w'\$ \vdash^*_{M'_n} \mathsf{Accept}.$$

Here w' belongs to the language L_n, and w' is obtained from $w = xabyabz$ by deleting

exactly one symbol. However, if a symbol is deleted from the prefix $xaby$, then the n-th symbol of w' is a b, and if a symbol is deleted from the suffix $yabz$, then the n-th last symbol of w' is an a. Thus, in either case we see that $w' \notin L_n$, a contradiction. This means that the accepting computation of M'_n on input w is just an accepting tail computation, that is, M'_n behaves essentially just like a DFA, which implies that it needs $\Omega(2^n)$ many states to check the condition $w_{m+1-n} = b$. □

Without giving further details, the descriptional complexity of these devices forms an interesting topic on its own. Here we will use it in further discussions on the practicability of the machines considered later.

2.3 Relation Classes and Transducers

Analogous to the characterizations of plain sets of words (i.e. languages) by certain types of automata or grammars, we show here what kinds of *binary relations* (i.e. sets of pairs of words) are characterized by different types of transducers or grammars with output. Further, we focus on the properties of these classes of relations, necessary for the topic of this work.

We already mentioned in the introduction of the present thesis that although the idea of computing relations instead of plain sets of words goes back to the early times of automata theory (e.g. [RS59]), there are only a few comprehensive studies on parts of that topic. In a certain sense, the latter might also be the reason for the lack of a uniform framework for studying the various types of relation classes. Here we establish such a framework according to our needs. For that, this section mainly concerns two special classes of relations including some of their proper subclasses, the class of *rational relations* and the class of *pushdown relations*. An extensive study of the class of rational relations can be found in Berstel [Ber79]. Further, hierarchy results on the subclasses of rational relations and also a study of pushdown relations is given by Choffrut and Culik II. [CI83]. Moreover, we mainly follow the notations given in the latter references. Last but not least we also recommend the books of Aho and Ullman [AU72], Eilenberg [Eil74] and more recent Sakarovitch [Sak09] for further reading. While the first one provides a more grammar based investigation of rational relations, pushdown relations and beyond, the other two offer an algebraic overview on rational relations and some of their subclasses.

We start with some general terms. According to the introduction we only deal with binary (word) relations, that is, a (binary) relation is a subset of the Cartesian product of two sets of words. A second possible description of such a relation is in the form of a mapping from the first set to the subsets of the second one, which we call a *transduction*. Obviously, these two descriptions coincide and we will use them equally. Since transductions are realized by transducers, it is natural to ask about the class of transductions that can be realized by a type of transducer, which is a main item of the present work. In the literature the notion of transductions, relations, mappings, and classes of relations/transductions computed by transducers seem often quite redundant or worse, confusing. For that we try to make these terms precise at this point.

Let T be a transducer (of any type) that realizes a transduction $T : \Sigma^* \to \mathfrak{P}(\Delta^*)$. Note that here T denotes both, the transducer and the transduction. Anyway, the meaning will be clear from the context. The transduction T is a (partial) function, where the domain of this (partial) function is defined as all words of Σ^* for which there is an output word (or a set of output words) from Δ^*, according to the transducer's definition. Hence, the relation that is defined by T is simply the graph of its transduction. Then, the *class of relations* that is realized by a type of transducer is the set of all transductions/relations defined by the different instances of transducers of that type. Clearly, this is a classification of the power of a type of transducer in analogy to types of automata that define language classes.

Another aspect, which helps to classify types of transducers, is in terms of *mappings*. Instead of considering the whole domain as the transducer's input, we may investigate only the mappings realized by taking only a subset of the domain. In other words, we take all words from a language of one language class and verify the resulting language. Later, this latter question is shortly discussed under the phrase "Preservation of Languages by Transducers" ([GR66, GR68], see p.69 "Properties").

Rational Relations and Finite Transducers

To understand the theory of rational relations and their characterization by finite transducers, the notion of recognizable and rational sets is mandatory. For further studies on recognizable and rational sets we refer to [Ber79] and [Eil74]. So we start with a little bit of algebra.

Definition 2.3.1 (Recognizable Sets)**.** *Let M be a monoid. A subset X of M is recognizable if there exist a finite monoid N, a morphism $\varphi : M \to N$ and a subset P of N such*

that $X = \varphi^{-1}(P)$. *The family of all recognizable subsets of* M *is denoted by* $\mathcal{R}ec(M)$.

Based on that definition one can show for an arbitrary alphabet Σ and the *finitely generated free monoid* Σ^* that $\mathcal{R}ec(\Sigma^*)$ is exactly the class of regular languages (over Σ). In general, the recognizable subsets of a finitely generated monoid are the sets that can be recognized by finite automata. Next we turn to rational sets.

Definition 2.3.2 (Rational Set)**.** *Let* M *be a monoid. The family* $\mathcal{R}at(M)$ *of rational subsets of* M *is the smallest set such that the following conditions hold:*

- *every finite subset of* M *belongs to* $\mathcal{R}at(M)$,
- *if* $X_1, X_2 \in \mathcal{R}at(M)$, *then also* $X_1 \cup X_2 \in \mathcal{R}at(M)$ *and* $X_1 \cdot X_2 \in \mathcal{R}at(M)$,
- *if* $X \in R$, *then also* $X^* \in \mathcal{R}at(M)$.

It is well known by Kleene's Theorem [Kle56] that in the case of a *finitely generated free monoid* Σ^*, the rational subsets coincide with the recognizable subsets and thus, $\mathcal{R}at(\Sigma^*) = \mathcal{R}ec(\Sigma^*) = \mathsf{REG}(\Sigma^*)$ holds. This is not true for an arbitrary monoid M. In fact, if M is a *finitely generated monoid*, then $\mathcal{R}ec(M) \subseteq \mathcal{R}at(M)$ (McKnight 1964, [McK64]). Let this become clearer by considering the following monoid $M = \Sigma^* \times \Delta^*$, where Σ and Δ are alphabets. Thus, M is defined as the Cartesian product of two finitely generated free monoids. Observe that M is finitely generated by $(\Sigma \times \{\varepsilon\}) \cup (\{\varepsilon\} \times \Delta)$, but not free. The reason for that is, an arbitrary element (x, y) can be generated in more than one way (e.g. $(x, y) = (x, \varepsilon) \cdot (\varepsilon, y) = (\varepsilon, y) \cdot (x, \varepsilon)$). In this case McKnights inclusion stated above is proper. For that, consider the relation $R \subseteq \{a\}^* \times \{b\}^*$ that is defined as $R = \{(a^n, b^n) \mid n \geq 0\}$. This relation is obviously rational (in the sense of the definition, by taking (a, ε), (ε, b) as generators of $\{a\}^* \times \{b\}^*$), but it is not recognizable. To substantiate this fact, we should mention a consequence of Mezei's Theorem (given e.g. in [Ber79]), that is, each recognizable subset of the product of two finitely generated free monoids can be expressed as the finite union of the Cartesian products of regular languages. In particular, if R is recognizable, then there exist finitely many regular languages $L_i \subseteq \{a\}^*$ and $L_j \subseteq \{b\}^*$, such that $R = \bigcup_{i,j} L_i \times L_j$. Clearly R cannot be defined in such a way. And in this sense finite automata cannot recognize all sets in $\mathcal{R}at(\Sigma^* \times \Delta^*)$.

However, before we turn to the machines that exactly characterize the sets in $\mathcal{R}at(\Sigma^* \times \Delta^*)$ we should adjust some of our previously mentioned notations. In the short introduction above we defined recognizable and rational sets over arbitrary monoids. Clearly, in the

following we are only interested in finitely generated free monoids, that is, alphabets, as a basis for our reflections. Therefore, we will rename the rational sets over the Cartesian product of two finitely generated free monoids in the next definition.

Definition 2.3.3 (Rational Relations)**.** *Let Σ and Δ be alphabets. A rational relation (*RAT *for short) over Σ and Δ is a rational subset of the monoid $\Sigma^* \times \Delta^*$.*

Actually we should call such relations rational *word* relations, but since it is clear in this context that we are only interested in words, we stick to the notion of rational relations. We may further mention that in the following "recognizable relations" play a subordinated role, as we are mainly interested in subclasses of rational relations here.

We continue by giving a first characterization of the class of rational relations in terms of regular languages, which can be proven as a direct consequence of the algebraic properties given above.

Theorem 2.3.4 ([Niv68])**.** *Let $R \subseteq \Sigma^* \times \Delta^*$ be a relation. R is rational if and only if there exist a finite alphabet Γ, a regular language $L \subseteq \Gamma^*$ and two morphisms $\varphi : \Gamma^* \to \Sigma^*$, $\psi : \Gamma^* \to \Delta^*$, such that*

$$R = \{(\varphi(w), \psi(w)) \mid w \in L\}.$$

This theorem will play an important role in the present work. However, we now turn to more machine based characterizations of rational relations with the aim to establish a hierarchy of relation classes below RAT.

In analogy to finite state automata we call the machines that characterize RAT *finite state transducers*. Here we follow the dynamic aspect of transductions (mentioned in the introduction) and define such a device as an NFA with an additional output tape.

Definition 2.3.5 (Finite State Transducer)**.** *A finite state transducer $T = (Q, \Sigma, \Delta, \delta, q_0, F)$ (*FST *for short) is a 6-tuple, where Q is a set of states, Σ is a finite input- and Δ a finite output-alphabet, q_0 is the initial state, F is the set of final states, and the transition function δ is defined as*

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathfrak{P}_{fin}(Q \times \Delta^*).$$

A *configuration* of an FST $T = (Q, \Sigma, \Delta, \delta, q_0, F)$ is a tuple (qu, v), where qu is the configuration of the underlying non-deterministic finite automaton and v is the output produced so

far. An *accepting computation* of T is described as follows, where $u_0, u_1, \ldots, u_k \in \Sigma \cup \{\varepsilon\}$, $v_0, v_1, \ldots, v_k \in \Delta^*$, $q_0, q_1, \ldots, q_k, q_{k+1} \in Q$ and $q_{k+1} \in F$:

$$(q_0u_0u_1 \ldots u_k, \varepsilon) \vdash_T (q_1u_1 \ldots u_k, v_0) \vdash_T (q_2u_2 \ldots u_k, v_0v_1) \vdash_T \cdots \vdash_T (q_{k+1}, v_0v_1 \ldots v_k).$$

Here the transitions are obviously defined as $(q_i, v_{i-1}) \in \delta(q_{i-1}, u_{i-1})$, for all $i = 1, \ldots, k+1$. Observe that although the length of the input and output depends on $k \in \mathbb{N}$ in our illustration, they actually are not linked, as $v_0, v_1, \ldots, v_k$ are words over Δ^* and $u_0, u_1, \ldots, u_k$ can possibly be empty. Further, note that the next step relation ($\vdash$) as well as its reflexive and transitive closure ($\vdash^*$) naturally extend to finite state transducers. Now the transduction $T : \Sigma^* \to \mathfrak{P}(\Delta^*)$ that is realized by the given FST is defined as:

$$T(u) = \{v \mid (q_0u, \varepsilon) \vdash_T^* (q, v) \text{ and } q \in F\}, \text{ for all } u \in \Sigma^*.$$

The relation computed by T is the graph of its transduction, that is, $\mathrm{Rel}(T) = \{(u, v) \in \Sigma^* \times \Delta^* \mid v \in T(u)\}$. Finally by $\mathcal{R}\mathrm{el}(\mathsf{FST})$ we denote the class of relations computed by an FST.

Example 2.3.6 ([Ber79])**.** Let $T = (\{q_0, q_1, \ldots, q_4\}, \{a\}, \{b, c\}, \delta, q_0, \{q_0, q_2, q_3\})$ be an FST, where δ is defined as shown in Figure 2.5. Thus, T realizes the transduction

$$T(a^n) = \begin{cases} b^n & , n \text{ even}, \\ c^n & , n \text{ odd}. \end{cases}$$

Hence, $\mathrm{Rel}(T) = \{(a^{2n}, b^{2n}), (a^{2n+1}, c^{2n+1}) \mid n \geq 0\}$.

We state the next result without citation as it can be found in nearly every textbook on formal language theory.

Theorem 2.3.7. $\mathcal{R}\mathrm{el}(\mathsf{FST}) = \mathsf{RAT}$.

We continue by defining several variants of finite state transducers, of which it is well known that they define relation classes within the rational relations. To avoid redundancy we use the basics given in Definition 2.3.5 and just explain how the machines differ.

A *generalized sequential machine* (GSM for short) is a finite state transducer $T = (Q, \Sigma, \Delta, \delta, q_0, F)$ for which $\delta : Q \times \Sigma \to \mathfrak{P}_{fin}(Q \times \Delta^*)$. Observe that a GSM is not allowed to perform ε-steps, that is, in each step it reads a single symbol. To keep

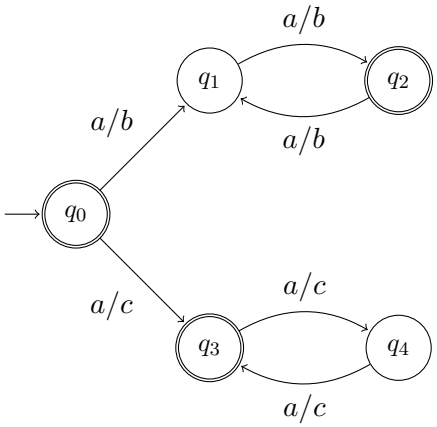

Figure 2.5: Example of a Finite State Transducer

technical things simple we require that $(\varepsilon, \varepsilon) \in \mathrm{Rel}(T)$ holds for each GSM T. By GSMRel we denote the class of all relations that are computed by GSMs.

A *deterministic generalized sequential machine* (dGSM for short) ([HU69], p. 172) is a generalized sequential machine $T = (Q, \Sigma, \Delta, \delta, q_0, F)$ for which δ is a partial function from $Q \times \Sigma$ into $\Delta^* \times Q$. [8] Hence, the relation $\mathrm{Rel}(T)$ is a partial function. By dGSMF we denote the class of all dGSM-functions.

A *sequential transducer* is a deterministic generalized sequential machine $T = (Q, \Sigma, \Delta, \delta, q_0, Q)$. Observe that all internal states of a sequential transducer are final. Clearly $\mathrm{Rel}(T)$ is again a (partial) function f and by SeqF we denote the class of all sequential functions.

A *subsequential transducer* consists of a pair $T_\varphi = (T, \varphi)$, where $T = (Q, \Sigma, \Delta, \delta, q_0, Q)$ is a sequential transducer and $\varphi : Q \to \Delta^*$ is a partial function. For $u \in \Sigma^*$, let $q_0 \cdot u \in Q$ denote the state that T reaches from its initial state q_0 on reading u. Then the relation $\mathrm{Rel}(T_\varphi)$ is defined as

$$\mathrm{Rel}(T_\varphi) = \{(u, z) \in \Sigma^* \times \Delta^* \mid \exists v \in \Delta^* : (u, v) \in \mathrm{Rel}(T) \text{ and } z = v\varphi(q_0 \cdot u)\}.$$

Obviously, $\mathrm{Rel}(T_\varphi)$ is a partial function. Observe that $(\varepsilon, \varphi(q_0)) \in \mathrm{Rel}(T_\varphi)$, if $\varphi(q_0)$ is

[8] Note that we should also define a deterministic version of a finite state transducer. However, this device would differ from a dGSM by having the ability to perform ε-steps. This, in fact leads to relations that are not functions ([AU72], p. 226).

defined, and that the word $\varphi(q_0)$ may well be non-empty. A partial function f is called *subsequential* if there exists a subsequential transducer T_φ that computes f. By SubSeqF we denote the class of all subsequential functions.

Finally, it is worth to introduce the class of *rational functions*, which is actually a superclass of all classes of functions described above. Additionally, later we will see that this class has a lot of nice properties, which are of great practical interest. However, literally rational functions are the rational relations that are actually functions, that is, *single valued rational relations* (in the sense of the Preliminaries). By RATF we denote the class of all rational functions. Next we will introduce two machine-like characterizations of this particular class. The most natural one is in terms of *unambiguous finite state transducers*. Informally an unambiguous finite state transducer is a finite state transducer, where for every input word there is at most one successful computation. Clearly such a transducer computes a (partial) function. Further, due to Eilenberg [Eil74] it can be shown that every rational function is computable by such a transducer. Actually, Berstel presented a proof of this result in [Ber79], where every rational function τ, with $\tau(\varepsilon)$ is either ε or undefined, yields an unambiguous finite state transducer that does not perform ε-steps. Thus, in terms of our notations *every rational function τ, where $\tau(\varepsilon) = \varepsilon$, is computable by an unambiguous* GSM. Note that it is easy to show directly that if we restrict the empty input as considered above, there is a unambiguous GSM for every unambiguous FST that realizes the same function. For that observe that a FST that computes a rational function will not perform loops of ε-steps, as this would contradict the property of being unambiguous. Hence, the output produced during a number of ε-steps can clearly be encoded in one step, which yields an unambiguous GSM[9]. Therefore, we have seen that ε-steps are only needed in a machine-like characterization, if we consider rational functions, where ε is mapped onto an arbitrary word over the output alphabet.

The latter characterization of the class of rational functions might not be very useful for applications, as in some sense being unambiguous is not a syntactical property[10]. However, Eilenberg [Eil74] introduced a special device for the class of rational functions, the bimachine.

[9] A confirmation of this statement and a few further information on rational functions and ε-free transducers can be found for instances in [RS96].

[10] Although it can be shown to be decidable [Sch75].

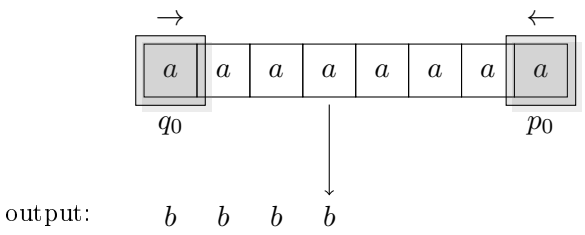

Figure 2.6: Sketch of the bimachine from Example 2.3.9.

Definition 2.3.8. *A bimachine* $B = (Q_1, Q_2, \Sigma, \Delta, \delta_1, \delta_2, q_0^{(1)}, q_0^{(2)}, \sigma)$ *is a 9-tuple, where* Q_1, Q_2 *are sets of states,* Σ *is a finite input- and* Δ *a finite output-alphabet,* $q_0^{(1)} \in Q_1$, $q_0^{(2)} \in Q_2$ *are two initial states,* $\delta_1 : Q_1 \times \Sigma \to Q_1$ *and* $\delta_2 : \Sigma \times Q_2 \to Q_2$ *are two (partial) transition functions and* $\sigma : Q_1 \times \Sigma \times Q_2 \to \Delta^*$ *is a (partial) output function.*

Without going into details we describe the modes of operation of this machine by an example taken from [Ber79].

Example 2.3.9. Let $B = (\{q_0, q_1\}, \{p_0, p_1\}, \{a\}, \{b, c\}, \delta_1, \delta_2, q_0, p_0, \sigma)$ be a bimachine, where δ_1 and δ_2 are defined as

$$\begin{aligned} \delta_1(q_0, a) &= q_1, \\ \delta_1(q_1, a) &= q_0, \end{aligned}$$

$$\begin{aligned} \delta_2(a, p_0) &= p_1, \\ \delta_2(a, p_1) &= p_0. \end{aligned}$$

Further, the output function σ is given by

$$\begin{aligned} \sigma(q_0, a, p_0) &= c, \\ \sigma(q_1, a, p_1) &= c, \\ \sigma(q_0, a, p_1) &= b, \\ \sigma(q_1, a, p_0) &= b. \end{aligned}$$

Hence, according to Figure 2.6 B scans the tape from left to right and right to left, simultaneously. The output is produced when both heads meet at the current input symbol[11], which moves from left to right across the given input. Then it is clear that B realizes the

[11] In Figure 2.6 the current input symbol is marked by the arrow.

same function as the finite state transducer in Example 2.3.6, that is,

$$B(a^n) = \begin{cases} b^n & , n \text{ even}, \\ c^n & , n \text{ odd}. \end{cases}$$

The definition and the example show that such a bimachine mirrors the computation of two finite transducers, in particular, a (left) sequential transducer and a right sequential transducer[12]. Furthermore, it is well known that each rational function τ, with $\tau(\varepsilon)$ is either ε or undefined, is a composition of a left sequential and a right sequential function. Consequently, Eilenberg ([Eil74], p.325) showed that a bimachine characterizes this class of rational functions.

A hierarchy of the various types of rational relations and functions can be found in the next subsection (Figure 2.7), where they are also compared with subclasses of pushdown relations.

Some properties of the relation classes introduced above can be found on page 69 in this subsection.

Pushdown Relations

As already mentioned in the introduction, there are only a few notable works on pushdown relations. Here [CI83] and [AU72] will serve as our main references. According to the extension of a finite state automaton to a transducer we start with the definition of a pushdown transducer, that is a pushdown automaton with an additional output tape.

Definition 2.3.10 (Pushdown Transducer)**.** *A pushdown transducer* $T = (Q, \Sigma, \Delta, \Gamma, \delta, q_0, Z_0, F)$ *(*PDT *for short) is an 8-tuple, where* Q *is a set of states,* Σ *is a finite input- ,* Δ *a finite output-alphabet and* Γ *a finite pushdown-alphabet,* q_0 *is the initial state,* F *is the set of final states, and* Z_0 *is the initial stack symbol. Finally* δ *is defined as*

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathfrak{P}_{fin}(Q \times \Gamma^* \times \Delta^*).$$

A *configuration* of $T = (Q, \Sigma, \Delta, \Gamma, \delta, q_0, Z_0, F)$ is a 3-tuple (qu, α, v), where (qu, α) is the configuration of the underlying non-deterministic pushdown automaton and v is the output

[12]Note that a right sequential transducer is simply a sequential transducer that scans the tape from right to left and produces its output in the same direction

produced so far. Again the next step relation ($\vdash$) as well as its reflexive and transitive closure ($\vdash^*$) naturally extends to pushdown transducers. For $(p, \beta, y) \in \delta(q, a, Z)$, we define a *computation step* of T as follows, where $p, q \in Q$, $Z \in \Gamma$, $\alpha, \beta \in \Gamma^*$, $x \in \Sigma^*$, $a \in \Sigma \cup \{\varepsilon\}$ and $v, y \in \Delta^*$:

$$(qax, \alpha Z, v) \vdash_T (px, \alpha\beta, vy).$$

Now the transduction $T : \Sigma^* \to \Delta^*$ that is realized by the given PDT is defined as:

$$T(u) = \{v \mid (q_0 u, Z_0, \varepsilon) \vdash_T^* (q, \alpha, v) \text{ for } \alpha \in \Gamma^* \text{ and } q \in F\}, \text{ for all } u \in \Sigma^*.$$

The relation computed by T is the graph of its transduction, that is, $\mathrm{Rel}(T) = \{(u, v) \in \Sigma^* \times \Delta^* \mid v \in T(u)\}$. Finally by PDR we denote the class of relations computed by PDT, the *pushdown relations*. Note, although it is possible to define this transducer such that they accept by empty stack instead of by final state, we here only use the above definition[13].

We continue by introducing two possible restrictions of the pushdown transducer. A pushdown transducer is called *unambiguous* (UPDT for short) if its underlying pushdown automaton is unambiguous ([Har78], p. 142). Clearly the relations computed by UPDTs are (partial) functions. By UPDF we denote the class of all *unambiguous pushdown functions*.

Further, we call a pushdown transducer $T = (Q, \Sigma, \Delta, \Gamma, \delta, q_0, Z_0, F)$ deterministic (DPDT for short), if δ is a (partial) function from $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ onto $Q \times \Gamma^* \times \Delta^*$ and $|\delta(q, a, Z) + \delta(q, \varepsilon, Z)| \leq 1$, for $q \in Q$, $a \in \Sigma$ and $Z \in \Gamma$. By DPDR we denote the class of *deterministic pushdown relations*. Observe that the latter class contains relations that are not functions. For instance, consider a DPDT that outputs symbols while performing a cycle of ε-steps on a final state. To overcome this minor issue, we additionally define two classes of functions. The *pushdown functions* (PDF for short), that is the class of pushdown relations that are functions, and accordingly the *deterministic pushdown functions* (DPDF).

Example 2.3.11. Let $T = (Q, \{a, b\}, \{a, b\}, \{A, B\}, \delta, q_0, Z_0, F)$ be a PDT with $Q = \{q_0, q_1, q_e\}$, $F = \{q_e\}$ and δ is defined as follows, where $x, y \in \Sigma$ and X, Y are their corresponding (for instance capital) symbols in Γ:

[13] Indeed it can be shown in analogy to automata that both definitions are equivalent.

$$
\begin{array}{lrcl}
(1) & \delta(q_0, x, Z_0) & = & (q_0, Z_0X, \varepsilon), \\
(2) & \delta(q_0, y, X) & = & (q_0, XY, \varepsilon), \\
(3) & \delta(q_0, \varepsilon, X) & = & (q_1, \varepsilon, x), \\
(4) & \delta(q_1, \varepsilon, X) & = & (q_1, \varepsilon, x), \\
(5) & \delta(q_1, \varepsilon, Z_0) & = & (q_e, Z_0, \varepsilon).
\end{array}
$$

It is easy to see that $T(w) = w^R$ for all $w \in \Sigma^+$, and $\mathrm{Rel}(T) = \{(w, w^R) \mid w \in \Sigma^+\}$. Note that T is actually an unambiguous PDT and thus, $T(w)$ is an unambiguous pushdown function.

According to [CI83] Figure 2.7 combines the various types of rational and pushdown relations. There, arrows denote proper inclusions, and classes that are not connected are incomparable. For that it is clear that the class GSMRel is a superclass of the class of dGSM-functions, and that it is incomparable to the classes of subsequential and rational functions with respect to inclusion. On the one hand, the reason for the latter is that GSMs can compute relations that are not functions, and on the other hand, a GSM cannot produce a non-empty output on empty input. Further, deterministic pushdown functions are incomparable to subsequential functions for the reason that a deterministic pushdown transducer is not able to "guess" the end of an input word without violating the property that $|\delta(q, a, Z) + \delta(q, \varepsilon, Z)| \leq 1$.

We continue by briefly introducing another way to define relations, namely by so called *syntax directed translation schemes*. Analogue to the notion of grammars for generating languages, a syntax directed translation scheme generates relations. An overview can be found in the book of Aho and Ullman [AU72] and we recommend [AU69, AHU69] for more details.

Definition 2.3.12 (Simple Syntax Directed Translation Scheme)**.** *A simple syntax directed translation scheme (*sSDTS *for short) is defined as a 5-tuple $D = (V, \Sigma, \Delta, P, S)$, where V is a finite set of non-terminals, Σ is a finite input- and Δ a finite output alphabet, P is a finite set of rules of the form $A \to \alpha, \beta$, where $\alpha \in (V \cup \Sigma)^*$, $\beta \in (V \cup \Delta)^*$ and the non-terminals in β occur in the same order as in α.*[14] *Finally S denotes the start symbol.*

Roughly speaking a sSDTS is a combination of two context-free grammars that are controlled both by the same non-terminals. Analogues to grammars for languages $\Rightarrow$ denotes

[14] This property is actually the reason why the defined syntax directed translation scheme is called simple.

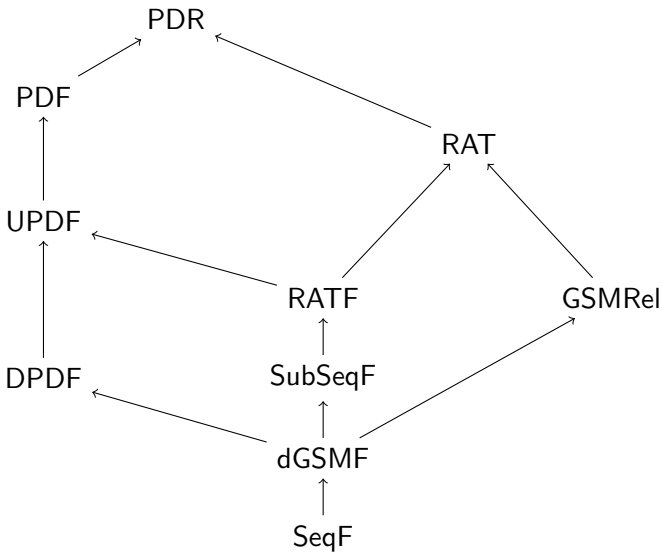

Figure 2.7: Inclusions between the various types of rational and pushdown relations.

the derivation relation and $\Rightarrow^*$ the reflexive and transitive closure. Without going into further details, the relation generated by a sSDTS $D = (V, \Sigma, \Delta, P, S)$ is defined as

$$\mathrm{Rel}(D) = \{(u, v) \mid (S, S) \Rightarrow_D^* (u, v)\}$$

and $\mathcal{R}\mathrm{el}(\mathsf{sSDTS})$ denotes the class of relations generated by syntax directed translation schemes. The following equivalence from the cited references causes our interest in syntax directed translation schemes:

$$\mathcal{R}\mathrm{el}(\mathsf{sSDTS}) = \mathcal{R}\mathrm{el}(\mathsf{PDT}) = \mathsf{PDR}.$$

At the beginning of this chapter we introduced the notions of Chomsky and Greibach normal forms for grammars. The next result extends these normal forms to sSDTS. To keep things simple we stick to the common names.

Proposition 2.3.13 ([AU69])**.** *A relation is a pushdown relation if and only if it is defined by an* sSDTS *in Chomsky (Greibach) normal form.*

Without going into details, it can be shown further that the latter result is also true for sSDTS in quadratic Greibach normal form ([AU69], p.327).

Finally note that if we use regular grammars instead of context-free ones, the class of relations of simple syntax directed translations schemes coincides with the rational relations.

Properties

The following properties are worth to mention in the context of rational and pushdown relations. Most of them will be frequently used to verify the performance of the types of transducers introduced in the following chapters. However, we start by returning to the notion of transducer mappings, which was briefly mentioned in the introduction of the present section. Instead of focusing on the relation classes that are computed by certain types of transducers, here we investigate which language classes are preserved under transducer mappings. However, to illustrate this point we provide two simple examples.

Example 2.3.14. Consider the relation $R = \{(w, w) \mid w \in \Sigma^*\}$. Obviously, R is a rational relation. An FST T for R can easily be described. Let $L \subseteq \Sigma^*$ be context free, then T maps L onto itself. Hence, we say that T preserves context-freeness.

A second example.

Example 2.3.15. Let $\tau : \Sigma^* \to \Sigma^*$ be the function that is defined as follows, where $n, m > 0$:

$$\tau(w) = \begin{cases} a^m b^n c^n & ; \text{ if } w = a^m b^n c^n, \\ \text{undefined} & ; \text{ else.} \end{cases}$$

It is easy to see that there is a PDT T for τ. However, if L is the context-free language $\{a^m b^m c^n \mid m, n > 0\}$, then $T(L) = \{a^m b^m c^m \mid m > 0\}$, which is well known to be not context free. Hence, T does not preserve context-freeness.

Generally the following result on finite state transducers can be shown (e.g. in [Ber79], p.58).

Proposition 2.3.16. *Each finite state transducer preserves regular and context-free languages.*

Note that the latter result does not hold for context-sensitive languages. This can be easily shown by applying a finite transducer mapping to the context-sensitive language

of valid computations of a particular Turing Machine such that (roughly speaking) all configurations except the initial one are deleted. In this sense a context-sensitive language is mapped onto a recursively enumerable one.

The situation for pushdown transducers is more complicated ([RS97], p.155). An immediate consequence of Example 2.3.15 is the following corollary.

Corollary 2.3.17. *The image of a context-free language under a pushdown transducer is not necessarily context free.*

In fact, it can be shown that for every recursively enumerable language L, there is a context-free language L' and a pushdown transducer T such that $T(L') = L$. This fact and the following two results can be found in [GR66, GR68].

Proposition 2.3.18. *The image of a regular language under a pushdown transducer is context free.*

Theorem 2.3.19. [15] *Let T be a* PDT*. If L is the context-free language accepted by the underlying* PDA*, then $T(L)$ is a context-free language.*

Recall in this context that the class of pushdown relations is defined as the class of all sets of pairs of words that are accepted by a pushdown transducer. Thus, when we talk about pushdown relations (transductions), we intend mappings from context-free onto context-free languages.

Finally, we turn to some *closure properties* and *decision problems* of the present transducers and relation classes. Admittedly, the investigation of such properties plays only a minor role in our further reflections. However, there are a few remarkable exceptions, namely closure under composition and decidability of equivalence and the so called type-checking problem (see Section 4.4 and 4.5). These properties are all known to be of major interest for practical purposes, such as natural language processing (e.g. [KK94]) or XML document transformation (e.g. [RS08]).

We start by briefly addressing the main closure properties of the class of rational relations. Here we refer to [Ber79] again for a general overview on closure properties of rational and recognizable sets and we should also mention a technical report of Roche and Schabe [RS96], where some nice constructive proofs for these properties are shown.[16]

[15] It is worth to mention that this result originally stems from the doctoral thesis "The Theory and Applications of Pushdown Store Machines" of R.J. Evey (1963).

[16] A first detailed reflection on these properties was given by Elgot and Mezei in [EM65].

According to the situation for regular languages we can immediately obtain the following closure properties as a consequence of the definition of rational sets (Definition 2.3.2).

Proposition 2.3.20. *The class of rational relations is closed under union, concatenation, and star-operation.*

In contrast to the situation for regular languages we have the following.

Proposition 2.3.21. *The class of rational relations is not closed under intersection and complement.*

For that consider the following two rational relations $R_1 = \{(a^n, b^n c^m) \mid n, m \geq 0\}$ and $R_2 = \{(a^n, b^m c^n) \mid n, m \geq 0\}$. Thus, $R_1 \cap R_2 = \{(a^n, b^n c^n) \mid n \geq 0\}$, which is clearly not rational (Proposition 2.3.16). It follows that RAT is also not closed under complement.

Most notable is the following result.

Theorem 2.3.22 ([EM65])**.** *The class of rational relations is closed under composition.*

Observe that the latter result can be shown constructively by simulating two transducers in parallel. Hence, this technique can easily be extended to all types of finite transducers previously defined (see Subsection 4.4). Thus, they are all closed under composition.

Concerning decision problems we should mention that most of the interesting questions are undecidable for RAT. In this context the equivalence problem for relations stands out and is defined as follows:

Instance: Given a class of relation $\mathcal{R}$ and two relations $R_1 \in \mathcal{R}$ and $R_2 \in \mathcal{R}$.

Question: Is $R_1 = R_2$?

Although it is undecidable for RAT, it becomes decidable for single valued rational relations (e.g. in [BH77]).

Theorem 2.3.23. *Equivalence is decidable for the class of rational functions.*

The latter result and the fact that being functional (as well as sequential and subsequential) is decidable (e.g. in [Sch75], [BH77], and [Cho77]) for rational relations causes the importance of rational functions and their subclasses in many applications.

Next we turn to *pushdown relations*. Clearly they are *closed under union*, while they are *not closed under composition*. For that consider the pushdown relations

$$R_1 = \{(a^n b^m c^m, a^n b^m c^m) \mid n, m \geq 0\} \text{ and } R_2 = \{(a^n b^n c^m, a^n b^n c^m) \mid n, m \geq 0\}.$$

Obviously

$$R_2 \circ R_1 = \{(a^n b^n c^n, a^n b^n c^n) \mid n \geq 0\},$$

which is not a pushdown relation (cf. Theorem 2.3.19).

Further, it is neither decidable if a pushdown relation is a function nor if two pushdown relations are equivalent. Both can easily be shown as consequences of undecidability results of context-free languages. Nevertheless it is remarkable that equivalence is decidable for deterministic pushdown functions [Sén99].

Chapter 3

Relations Associated to Restarting Automata and to Parallel Communicating Systems

Here we present two quite direct ways of assigning a relation to a restarting automaton. The first approach, which was originally suggested by Otto in [Ott10], concerns a more algebraic view on relations. A relation can be associated to an arbitrary automaton by splitting its alphabet into an input and output part. Then, by using a projection onto the input and output alphabets, the accepted language is transformed into a relation.

In the second part of this chapter we introduce Parallel Communicating Systems of Restarting Automata that consist of only two components. Motivated by the fact that relations are the connection of an input language and an output language, we use one component to accept the input and the other to accept the output. In this way a relation is defined. However, in older papers on computing relations instead of plain languages often two tape automata were considered (e.g. in [RS59]). Thus, this latter approach can be seen as an extension of the definition of relations in terms of two tape automata.

3.1 Input/Output-Relations and Proper-Relations

To understand the reason why we associate a relation to a restarting automaton by applying projections to its accepted language, we have to recall the linguistic motivation of these

types of automata. In Section 2.2 it was shown that restarting automata are basically the computer science equivalent to the linguistic technique of Analysis by Reduction. In the enriched form of Analysis by Reduction an input sentence is expected, where every word is annotated with a finite number of auxiliary symbols[17]. These annotations are used to hopefully derive a disambiguated form of the input. Then this sentence is repeatedly simplified with respect to the annotations until a basic form is reached. When taking a closer look on the definition of restarting automata, one can see that not only the simplification process is implemented. Also the different languages that appear in the analyzing process of natural languages can be associated to these automata. The reason for that is, if one considers auxiliary symbols in addition to the input alphabet, then the language, which is actually accepted, consists of words defined over the input and auxiliary alphabets. Thus, as restarting automata are forgetting machines, they cannot distinguish in their computation between original input words and words that appear in one of their restarting configurations.

Back to the motivation this means that the "actual" language of such an automaton corresponds to the language annotated by auxiliary symbols, which is used in the linguistic background. Further, we had already mentioned that the focus of this work is not only on the correctness of such a sentence but also on the information that can be extracted from it. From this point of view, this special language of words annotated by linguistic properties carries two types of information, that is, the actual input part itself and the annotations, which give a deeper insight into the sentence structure. Hence, it seems quite natural to associate a binary relation to this language by projecting onto input and auxiliary (output) part of every word.

3.1.1 Definitions and Examples

After the previous informal explanation, we start by presenting the formal basis of the following chapter. Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be a restarting automaton with tape alphabet Γ, which contains the input alphabet Σ. Here we call a word defined over the tape alphabet *sentential form*[18]. Recall that such a word (sentential form) $w \in \Gamma^*$ is accepted by M, if there is an accepting computation which starts from the restarting configuration $q_0¢w\$$. Hence, we can associate three different languages with M. By $L_C(M)$ we denote

[17] Note again that these auxiliary symbols have their origins in the different phases of analyzing a sentence of a natural language according to Section 2.2 (see "Analysis by Reduction").

[18] Note that sentential forms might consist of input and auxiliary symbols.

the language consisting of all sentential forms accepted by M; $L_{\mathrm{C}}(M)$ is the *characteristic language* of M, while the set $L(M) = L_{\mathrm{C}}(M) \cap \Sigma^*$ of all input sentences accepted by M is the *input language recognized* by M. Finally, $L_{\mathrm{P}}(M) = \mathsf{Pr}^{\Sigma}(L_{\mathrm{C}}(M))$ is the *proper language* of M, where $\mathsf{Pr}^{\Sigma} : \Gamma^* \to \Sigma^*$ denotes the *projection* that is defined as $a \mapsto a$ for each $a \in \Sigma$ and $A \mapsto \varepsilon$ for each $A \in \Gamma \backslash \Sigma$. Observe that the proper language forms an interesting research topic on its own, as it can be interpreted as the set of words for which there is a correct annotation in the characteristic language, which are (in principle) disambiguated sentences in terms of the linguistic motivation.

Based on the latter we define two different relations, where the restarting automaton above is extended by an output alphabet.

Definition 3.1.1. *Let $M = (Q, \Sigma', \Gamma, \mathfrak{c}, \$, q_0, k, \delta)$ be any type of restarting automaton with tape alphabet Γ and input alphabet Σ', which now contains the input alphabet Σ and the* output alphabet Δ*, such that $\Sigma' = \Sigma \cup \Delta$. Further we assume that Σ and Δ are disjoint. With M we associate two relations, where $\mathsf{Pr}^{\Sigma} : \Gamma^* \to \Sigma^*$ denotes the* projection *that is defined as $a \mapsto a$ for each $a \in \Sigma$ and $A \mapsto \varepsilon$ for each $A \in \Gamma \backslash \Sigma$:*

$$\begin{aligned} \mathrm{Rel}_{\mathrm{io}}(M) &= \{(u, v) \in \Sigma^* \times \Delta^* \mid \exists w \in L(M) : u = \mathsf{Pr}^{\Sigma}(w) \text{ and } v = \mathsf{Pr}^{\Delta}(w)\}, \\ \mathrm{Rel}_{\mathrm{P}}(M) &= \{(u, v) \in \Sigma^* \times \Delta^* \mid \exists w \in L_{\mathrm{C}}(M) : u = \mathsf{Pr}^{\Sigma}(w) \text{ and } v = \mathsf{Pr}^{\Delta}(w)\}. \end{aligned}$$

Here $\mathrm{Rel}_{\mathrm{io}}(M)$ is the input/output-relation *of M, and $\mathrm{Rel}_{\mathrm{P}}(M)$ is the* proper-relation *of M. By $\mathcal{R}\mathrm{el}_{\mathrm{io}}$ ($\mathcal{R}\mathrm{el}_{\mathrm{P}}$) we denote the class of input/output-relations (proper-relations, respectively) defined by a specified type of automata.*

Example 3.1.2. Let $M = (Q, \{a, b, c\}, \{a, b, c, \#\}, \mathfrak{c}, \$, q_0, 3, \delta\}$ be a det-mon-RRWW-automaton, where the output alphabet is defined as $\Delta = \{c\}$. The transition function δ is defined as follows:

$$\begin{aligned} (1)\quad \delta(q_0, \mathfrak{c}aa) &= (q_0, \mathsf{MVR}), \\ (2)\quad \delta(q_0, \mathfrak{c}ab) &= (q_0, \mathsf{MVR}), \\ (3)\quad \delta(q_0, aaa) &= (q_0, \mathsf{MVR}), \\ (4)\quad \delta(q_0, aab) &= (q_0, \mathsf{MVR}), \\ (5)\quad \delta(q_0, abc) &= (q_t, \varepsilon), \\ (6)\quad \delta(q_0, \mathfrak{c}\$) &= (\mathsf{Accept}), \end{aligned}$$

(7)	$\delta(q_0, ¢\#a)$	$=$	(q_1, MVR),	(15)	$\delta(q_2, a\#b)$	$=$	(q_4, MVR),
(8)	$\delta(q_0, ¢\#\#)$	$=$	(q_3, MVR),	(16)	$\delta(q_3, \#\#b)$	$=$	(q_4, MVR),
(9)	$\delta(q_0, ¢\#b)$	$=$	(q_4, MVR),	(17)	$\delta(q_4, abc)$	$=$	(q_t, ε),
(10)	$\delta(q_1, \#a\#)$	$=$	(q_2, MVR),	(18)	$\delta(q_4, \#bc)$	$=$	(q_t, ε),
(11)	$\delta(q_2, a\#\#)$	$=$	(q_3, MVR),	(19)	$\delta(q_t, bcb)$	$=$	$(q_{t'}, \mathsf{MVR})$,
(12)	$\delta(q_3, \#\#\#)$	$=$	(q_3, MVR),	(20)	$\delta(q_{t'}, cbc)$	$=$	(q_t, MVR),
(13)	$\delta(q_1, \#ab)$	$=$	(q_4, MVR),	(21)	$\delta(q_t, bc\$)$	$=$	$(\mathsf{Restart})$,
(14)	$\delta(q_2, a\#a)$	$=$	(q_1, MVR),	(22)	$\delta(q_t, \$)$	$=$	$(\mathsf{Restart})$.

Mainly the transition function of M leads to two different computations. The first one is defined by the left column (transitions (1) - (6)). Obviously by the first five transitions M checks whether the word on the tape is of the form a^+bc, then one substring abc is deleted, and finally the transitions (19) - (22) handle the right computation, where M verifies that the until then unseen part of the tape corresponds to $(bc)^*$. Thus, it is clear that with this part of the transition function the automaton accepts the language $\{a^n(bc)^n \mid n \geq 0\}$.

The second possible computation of M starts by using one of the transitions (7) - (9). Hence, on its tape the automaton expects a prefix of the form $(\#a)^*$ (transitions (7),(10) and (14)) that is possibly followed by a number of #-symbols (transitions (8), (11) and (12)). Now M proceeds as in the first computation, that is, every substring of the form abc or $\#bc$ is deleted (transitions (9), (13) and (15) - (18)) and again the unseen part is verified (transitions (19) - (22)). Observe that # is an auxiliary symbol and that both possible computations described above are disjunctive, which means that there is no possibility to mix up both parts, such that words are accepted by a shuffle of both transitions. Thus, over all the characteristic language of M is

$$L_C(M) = \{(\#a)^n\#^k(bc)^{2n+k} \mid n, k \geq 0\} \cup \{a^n(bc)^n \mid n \geq 0\},$$

the input language is $L(M) = L_C(M) \cap \Sigma^* = \{a^n(bc)^n \mid n \geq 0\}$ and the proper language is $L_p(M) = \mathsf{Pr}^{\Sigma}(L_C(M)) = \{a^n(bc)^m \mid n \geq 0, m \geq 2n\} \cup \{a^n(bc)^n \mid n \geq 0\}$. Then, by definition of input/output- and proper-relation

$$\mathrm{Rel}_P(M) = \{(a^nb^n, c^n), (a^nb^m, c^m) \mid n \geq 0, m \geq 2n\}$$

and

$$\mathrm{Rel}_{io}(M) = \{(a^nb^n, c^n) \mid n \geq 0\}.$$

3.1.2 Input/Output- and Proper-Relations of Monotone Restarting Automata

To derive some immediate results from Definition 3.1.1, here we recall a similar concept taken from [AU72]. This is in fact only a generalization of Theorem 2.3.4.

Definition 3.1.3. *A language L characterizes a relation R if there exist two homomorphism h_1 and h_2, such that*

$$R = \{(h_1(w), h_2(w)) \mid w \in L\}.$$

Definition 3.1.4. *A language $L \subseteq (\Sigma \cup \Delta')^*$ strongly characterizes a relation $R \subseteq \Sigma^* \times \Delta^*$ if*

(1) $\Sigma \cap \Delta' = \emptyset$ *and*

(2) $R = \{(h_1(w), h_2(w)) \mid w \in L\}$, *where*

a) $h_1(a) = a$ *for all* $a \in \Sigma$ *and* $h_1(b) = \varepsilon$ *for all* $b \in \Delta'$,

b) $h_2(a) = \varepsilon$ *for all* $a \in \Sigma$ *and* h_2 *is a copy isomorphism between* Δ' *and* Δ, *that is,* $h_2(b) \in \Delta$ *for all* $b \in \Delta'$ *and* $h_2(b) = h_2(b')$ *implies that* $b = b'$.

Also in [AU69] and [AU72] the latter definition was used to characterize the pushdown relations by considering context-free languages as their characterizing languages and regular languages as the characterizing languages of rational relations.

Theorem 3.1.5 ([AU69]). *A relation R is a pushdown relation* PDR *if and only if R is strongly characterized by a context-free language.*

We state two immediate consequences of the above definitions and Theorem 2.3.4 here. The first one concerns relations strongly characterized by regular languages.

Corollary 3.1.6 ([AU72]). *A relation R is a rational relation if and only if R is strongly characterized by a regular language.*

Secondly, we should also mention the following result of Nivat, which is a corollary of Theorem 2.3.4. It shows the similarities between the different definitions of characterizing relations in terms of morphisms.

Corollary 3.1.7. *A relation $R \subseteq \Sigma^* \times \Delta^*$ with $\Sigma \cap \Delta = \emptyset$ is rational (*RAT*) if and only if there exists a regular language $L \subseteq (\Sigma \cup \Delta)^*$ such that $R = \{(\mathsf{Pr}^{\Sigma}(w), \mathsf{Pr}^{\Delta}(w)) \mid w \in L\}$.*

We continue by combining the latter definitions and theorems with the definition of input/output-relations. First of all it is clear that the definition of input/output-relations, when applied to restarting automata that characterize the regular languages, leads to the same result as stated in Corollary 3.1.7. Thus, all these automata compute exactly the rational relations with the restriction on disjoint input and output alphabets.

Secondly, Definition 3.1.4 avoids this restriction ("disjoint alphabets") by using one projection and an isomorphism instead of two projections to map onto input and output. This leads to exact characterizations of both classes, the pushdown and rational relations (cf. Theorem 3.1.5 and Corollary 3.1.6). However, the difference between Definition 3.1.1 and Definition 3.1.4 is marginal. It can easily be compensated for by applying a simple isomorphism to the output alphabet. Hence, here we do not adjust our definitions as it is clear that it is not of importance that the input and output alphabets are disjoint.

Then, as mentioned already, the following results are immediate consequences of the latter theorems. For that recall from Section 2.2 the types of restarting automata that characterize the regular languages and context-free languages.

Proposition 3.1.8.

$$\mathsf{RAT} = \begin{cases} \mathcal{R}\mathrm{el}_{io}(\mathsf{det\text{-}R}(1)) \\ \mathcal{R}\mathrm{el}_{io}(\mathsf{mon\text{-}R}(1)) \\ \mathcal{R}\mathrm{el}_{io}(\mathsf{R}(1)) \\ \mathcal{R}\mathrm{el}_{io}(\mathsf{det\text{-}RR}(1)) \\ \mathcal{R}\mathrm{el}_{io}(\mathsf{det\text{-}mon\text{-}nf\text{-}R}(1)) \\ \mathcal{R}\mathrm{el}_{io}(\mathsf{mon\text{-}nf\text{-}R}(1)) \\ \mathcal{R}\mathrm{el}_{io}(\mathsf{det\text{-}mon\text{-}nf\text{-}RR}(1)) \end{cases}$$

Proposition 3.1.9.

$$\mathsf{PDR} = \begin{cases} \mathcal{R}\mathrm{el}_{io}(\mathsf{mon\text{-}RWW}) \\ \mathcal{R}\mathrm{el}_{io}(\mathsf{mon\text{-}RRWW}) \end{cases}$$

Beside these equivalences, also the following inclusion is clear.

Proposition 3.1.10 ([AU69])**.** *Every deterministic pushdown relation (*DPDR*) is strongly characterized by a deterministic context-free language.*

Actually this inclusion is proper for the following reason (see [AU69]):

Lemma 3.1.11. *There is a relation that is strongly characterized by a deterministic context-free language, but that is not a deterministic pushdown relation.*

Proof. Let $L_{pal} = \{wcw^R \mid w \in \{a,b\}^*\}$, that is, it is a well-known deterministic context-free language. Then it is clear that the relation $R_{pal} = \{(ww^R, c) \mid w \in \{a,b\}^*\}$ is strongly characterized by L_{pal}. It remains to show that $R_{pal} \notin \mathsf{DPDR}$. Recall that a relation is in DPDR if there exists a deterministic pushdown transducer that defines this particular relation. Hence, for our example this deterministic transducer has to accept inputs of the form ww^R and produce the output c. Obviously the language $L' = \{ww^R \mid w \in \{a,b\}^*\}$ is not accepted by any deterministic pushdown automaton. Thus, a deterministic pushdown transducer is not able to define the relation R. □

Proposition 3.1.12.

$$\mathsf{DPDR} \subsetneq \left\{ \begin{array}{c} \mathcal{R}\mathrm{el}_{io}(\mathsf{det\text{-}mon\text{-}R}) \\ \mathcal{R}\mathrm{el}_{io}(\mathsf{det\text{-}mon\text{-}RW}) \\ \mathcal{R}\mathrm{el}_{io}(\mathsf{det\text{-}mon\text{-}RWW}) \\ \mathcal{R}\mathrm{el}_{io}(\mathsf{det\text{-}mon\text{-}RR}) \\ \mathcal{R}\mathrm{el}_{io}(\mathsf{det\text{-}mon\text{-}RRW}) \\ \mathcal{R}\mathrm{el}_{io}(\mathsf{det\text{-}mon\text{-}RRWW}) \end{array} \right\} \subsetneq \mathsf{PDR}$$

Proof. The first proper inclusion is the consequence of Proposition 3.1.10 and Lemma 3.1.11. For the second inclusion consider the pushdown relation $R = \{(a^n b^n, \varepsilon), (a^n b^m, \varepsilon) \mid n \geq 1, m > 2n\}$. Let M be det-mon-RRWW-automaton such that $R = \mathrm{Rel}_{io}(M)$. Hence, it is clear that the accepted language of M must be $L(M) = \{a^n b^n, a^n b^m \mid n \geq 1, m > 2n\}$. As it is well known that this language is not in DCFL and thus it is not accepted by any det-mon-RRWW-automaton, it follows that there is no such automaton to compute R. □

Obviously in this way we can also define new classes of relations, e.g. the class of Church-Rosser relations by input/output relations of det-RWW- and det-RRWW-automata.

Next let us turn to the so called proper relations. In contrast to the situation above, here we restate a somewhat surprising result of Otto [Ott10]. For that, we need to mention that obviously the concept of characteristic languages and proper relations extends to all kinds of automata.

Theorem 3.1.13 ([Ott10]). $\mathcal{R}\mathrm{el}_P(\mathsf{DPDA}) = \mathsf{PDR}$

Thus, if we consider additional auxiliary symbols in the definition of a DPDA, we gain some power in terms of proper relations. For example, in the proof of the previous theorem the auxiliary symbols are used to encode a computation of a pushdown transducer on the tape of a DPDA. The next corollary is an immediate consequence of the latter result.

Corollary 3.1.14.

$$\mathsf{PDR} = \begin{cases} \mathcal{R}\mathrm{el}_P(\mathsf{det\text{-}mon\text{-}R}) \\ \mathcal{R}\mathrm{el}_P(\mathsf{det\text{-}mon\text{-}RW}) \\ \mathcal{R}\mathrm{el}_P(\mathsf{det\text{-}mon\text{-}RWW}) \\ \mathcal{R}\mathrm{el}_P(\mathsf{det\text{-}mon\text{-}RR}) \\ \mathcal{R}\mathrm{el}_P(\mathsf{det\text{-}mon\text{-}RRW}) \\ \mathcal{R}\mathrm{el}_P(\mathsf{det\text{-}mon\text{-}RRWW}) \end{cases}$$

Proof. By Theorem 3.1.13 the pushdown relations coincide with the class of proper relations of deterministic pushdown automata.

Further, according to Section 2.2 a language is deterministic context free if and only if there is a deterministic and monotone restarting automaton that accepts this language.

More formally, let $L_C(P)$ be the characteristic language of a DPDA P. $L_C(P)$ is clearly a deterministic context-free language. Then $\mathrm{Rel}_P(P)$ is a pushdown relation, derived by projecting $L_C(P)$ onto a distinguished input and output alphabet. Hence, it is clear that there is a deterministic and monotone restarting automaton M with $L_C(M) = L_C(P)$, and therefore, $\mathrm{Rel}_P(M) = \mathrm{Rel}_P(P)$. Clearly the converse direction can be treated the same. Hence, it is clear that a relation is a pushdown relation if and only if it is computed by a deterministic and monotone restarting automaton. □

3.2 Parallel Communicating Systems of Restarting Automata

Here we want to briefly present another way to associate relations to restarting automata. Instead of computing input and output by a single restarting automaton, it appears to be more natural to compute it by a pair of restarting automata that have the ability to communicate with each other. Hence, the first automaton serves as the acceptor of the

input language and the second one as the acceptor of the output language. Further, the communication relates particular words of both languages.

In this context parallel communicating restarting automata as language accepting devices were first introduced by Vollweiler and Otto and published in [VO12].

3.2.1 Definition

To formalize this idea, we introduce the so-called *parallel communicating system of restarting automata* (or PC-RRWW-system, for short). Obviously, every type of restarting automaton may serve as a component of such a system, however, as this chapter only presents an introduction that may lead to further investigations, we mainly use det-mon-RRWW-automata as the components of our systems.

Thus, a det-mon-PC-RRWW-system consists of a pair $\mathcal{M} = (M_1, M_2)$ of det-mon-RRWW-automata $M_i = (Q_i, \Sigma_i, \Gamma_i, ¢, \$, q_0^{(i)}, k, \delta_i)$ $(1 \leq i \leq 2)$. Here it is required that, for each $i \in \{1, 2\}$, the set of states Q_i of M_i contains finite subsets Q_i^{req} of *request states* of the form (q, req), Q_i^{res} of *response states* of the form $(q, \text{res}(l))$, Q_i^{rec} of *receive states* of the form $(q, \text{rec}(l))$, and Q_i^{ack} of *acknowledge states* of the form $(q, \text{ack}(l))$. Further, in addition to the move-right, rewrite, and restart steps, M_1 and M_2 have so-called *communication steps*.

A *configuration* of $\mathcal{M}$ consists of a pair $K = (k_1, k_2)$, where k_i is a configuration of M_i, $i = 1, 2$. For $w_1 \in \Sigma_1^*$ and $w_2 \in \Sigma_2^*$, the *initial configuration* on input (w_1, w_2) is $K_{\text{in}}(w_1, w_2) = (q_0^{(1)} ¢ w_1 \$, q_0^{(2)} ¢ w_2 \$)$. The *single-step computation relation* $(k_1, k_2) \vdash_{\mathcal{M}} (k_1', k_2')$ consists of *local steps* and *communication steps*:

$$
\begin{array}{lll}
\text{(Communication 1)} & \text{if} & k_1 = u_1(q_1, \text{res}(l))v_1 \text{ and } k_2 = u_2(q_2, \text{req})v_2, \text{ then} \\
 & & k_1' = u_1(q_1, \text{ack}(l))v_1 \text{ and } k_2' = u_2(q_2, \text{rec}(l))v_2; \\
\text{(Communication 2)} & \text{if} & k_1 = u_1(q_1, \text{req})v_1 \text{ and } k_2 = u_2(q_2, \text{res}(l))v_2, \text{ then} \\
 & & k_1' = u_1(q_1, \text{rec}(l))v_1 \text{ and } k_2' = u_2(q_2, \text{ack}(l))v_2.
\end{array}
$$

In all other cases M_1 and M_2 just perform local computation steps, independent of each other. If one of them is in a request or response state, but the other is not (yet) in the corresponding response or request state, respectively, then the latter automaton keeps on performing local steps until a communication step is enabled. Should this never happen, then the computation of $\mathcal{M}$ fails. Once one of M_1 and M_2 has accepted, the other automa-

ton keeps on performing local steps until it either gets stuck, in which case the computation fails, or until it also accepts, in which case the computation of $\mathcal{M}$ succeeds.

According to Subsection 3.1 we again associate different types of relations with a det-mon-PC-RRWW-system. Hence,

$$\mathrm{Rel}_{\mathrm{C}}(\mathcal{M}) = \{(w_1, w_2) \in \Gamma_1^* \times \Gamma_2^* \mid K_{\mathrm{in}}(w_1, w_2) \vdash_{\mathcal{M}}^* (\mathsf{Accept}, \mathsf{Accept})\}$$

is the *characteristic relation* of $\mathcal{M}$,

$$\mathrm{Rel}_{\mathrm{io}}(\mathcal{M}) = \mathrm{Rel}_{\mathrm{C}}(\mathcal{M}) \cap (\Sigma_1^* \times \Sigma_2^*) = \{(w_1, w_2) \in \Sigma_1^* \times \Sigma_2^* \mid K_{\mathrm{in}}(w_1, w_2) \vdash_{\mathcal{M}}^* (\mathsf{Accept}, \mathsf{Accept})\}$$

is the *input/output-relation* of $\mathcal{M}$, and

$$\mathrm{Rel}_{\mathrm{P}}(\mathcal{M}) = \{(\mathsf{Pr}^{\Sigma_1}(w_1), \mathsf{Pr}^{\Sigma_2}(w_2)) \mid (w_1, w_2) \in \mathrm{Rel}_{\mathrm{C}}(\mathcal{M})\}$$

is the *proper-relation* of $\mathcal{M}$.

To illustrate the definition above we start our reflections on the power of these systems with a short example.

Example 3.2.1. Let $\mathcal{M} = (M_1, M_2)$ be a det-mon-PC-RRWW-system, where $M_1 = (Q_1, \{a, b\}, \{a, b, [aa], [ab], [ba], [bb]\}, \mathfrak{c}, \$, q_0, 3, \delta_1)$ and $M_2 = (Q_2, \{a, b\}, \{a, b\}, \mathfrak{c}, \$, p_0, 1, \delta_2)$ and the transition functions of $\mathcal{M}$ are defined as follows, where $x_1, x_2, x_3, x_4 \in \{a, b\}$:

$$\begin{array}{lll}
M_1: & \delta_1(q_0, \mathfrak{c}x_1x_2) & = (q_0, \mathsf{MVR}), \\
& \delta_1(q_0, x_1x_2x_3) & = (q_0, req), \\
& \delta_1(q_0, x_1x_2\$) & = (q_0, req), \\
& \delta_1(q_0, \mathfrak{c}x_1[x_2x_3]) & = (q_0, \mathsf{MVR}), \\
& \delta_1(q_0, x_1x_2[x_3x_4]) & = (q_0, req), \\
& \delta_1(q_0, x_1[x_2x_3]\$) & = (q_0, req), \\
& \delta_1((q_0, rec(x_1)), x_1x_2x_3) & = (q_0, \mathsf{MVR}), \\
& \delta_1((q_0, rec(x_1)), x_1x_2[x_3x_4]) & = (q_0, \mathsf{MVR}), \\
& \delta_1((q_0, rec(x_1)), x_1x_2\$) & = (q_1, req), \\
& \delta_1((q_0, rec(x_1)), x_1[x_2x_3]\$) & = (q', req), \\
& \delta_1((q_1, rec(x_2)), x_1x_2\$) & = (q_R, [x_1x_2]\$), \\
& \delta_1((q', rec(x_2)), x_1[x_2x_3]\$) & = (q'', req),
\end{array}$$

$$\begin{array}{lll} \delta_1((q'', rec(x_3)), x_1[x_2x_3]\$) & = & (q_\$, req), \\ \delta_1((q_\$, rec(\$)), x_1[x_2x_3]\$) & = & (\mathsf{Accept}), \\ \delta_1(q_R, \$) & = & (\mathsf{Restart}). \end{array}$$

$$\begin{array}{lll} M_2: \quad \delta_2(p_0, \¢) & = & (p_0, \mathsf{MVR}), \\ \delta_2(p_0, x_1) & = & (p_0, res(x_1)), \\ \delta_2((p_0, ack(x_1)), x_1) & = & (p_0, \mathsf{MVR}), \\ \delta_2(p_0, \$) & = & (p_0, res(\$)), \\ \delta_2((p_0, ack(\$)), \$) & = & (\mathsf{Accept}). \end{array}$$

Thus, the first and the second component of $\mathcal{M}$ scan their tapes simultaneously letter by letter, while each symbol of M_1 is compared with the corresponding symbol of M_2. Hence, during the first cycle it is clear that the head of both machines is over the same position at the same "time". While M_1 reaches the \$-symbol, it restarts and the process starts anew. Observe that M_2 only executes a tail computation. Based on the transitions of $\mathcal{M}$ it follows that

$$\mathrm{Rel}_{io}(\mathcal{M}) = R_{copy} = \{(w, ww) \mid w \in \{a, b\}^* \text{ and } |w| \geq 3\}.$$

More formally, on input $(w_1, w_2) = (abba, abbaabba)$ $\mathcal{M}$ proceeds as follows:

$$\begin{array}{lll} (q_0¢abba\$, p_0¢abbaabba\$) & \vdash_{\mathcal{M}} & (¢\, q_0abba\$, ¢\, p_0abbaabba\$) \\ & \vdash_{\mathcal{M}} & (¢\, (q_0, req)abba\$, ¢\, (p_0, res(a))abbaabba\$) \\ & \vdash_{\mathcal{M}} & (¢\, (q_0, rec(a))abba\$, ¢\, (p_0, ack(a))abbaabba\$) \\ & \vdots & \vdots \\ & \vdash_{\mathcal{M}} & (¢\, abq_0ba\$, ¢\, abp_0baabba\$) \\ & \vdash_{\mathcal{M}} & (¢\, ab(q_0, req)ba\$, ¢\, ab(p_0, res(b))baabba\$) \\ & \vdash_{\mathcal{M}} & (¢\, ab(q_0, rec(b))ba\$, ¢\, ab(p_0, ack(b))baabba\$) \\ & \vdash_{\mathcal{M}} & (¢\, ab(q_1, req)ba\$, ¢\, abbp_0aabba\$) \\ & \vdash_{\mathcal{M}} & (¢\, ab(q_1, req)ba\$, ¢\, abb(p_0, res(a))aabba\$) \\ & \vdash_{\mathcal{M}} & (¢\, ab(q_1, rec(a))ba\$, ¢\, abb(p_0, ack(a))aabba\$) \\ & \vdash_{\mathcal{M}} & (¢\, ab[ba]q_R\$, ¢\, abbap_0abba\$) \\ & \vdash_{\mathcal{M}} & (q_0¢ab[ba]\$, ¢\, abba(p_0, res(a))abba\$) \\ & \vdash_{\mathcal{M}} & (¢\, q_0ab[ba]\$, ¢\, abba(p_0, res(a))abba\$) \\ & \vdash_{\mathcal{M}} & (¢\, (q_0, req)ab[ba]\$, ¢\, abba(p_0, res(a))abba\$) \\ & \vdots & \vdots \end{array}$$

$$
\begin{array}{ll}
\vdots & \vdots \\
\vdash_{\mathcal{M}} & (\mathrm{¢}\, a(q_0, req) b[ba]\$, \mathrm{¢}\, abbaa(p_0, res(b))bba\$) \\
\vdash_{\mathcal{M}} & (\mathrm{¢}\, a(q_0, rec(b)) b[ba]\$, \mathrm{¢}\, abbaa(p_0, ack(b))bba\$) \\
\vdash_{\mathcal{M}} & (\mathrm{¢}\, a(q', req) b[ba]\$, \mathrm{¢}\, abbaabp_0ba\$) \\
\vdash_{\mathcal{M}} & (\mathrm{¢}\, a(q', req) b[ba]\$, \mathrm{¢}\, abbaab(p_0, res(b))ba\$) \\
\vdash_{\mathcal{M}} & (\mathrm{¢}\, a(q', rec(b)) b[ba]\$, \mathrm{¢}\, abbaab(p_0, ack(b))ba\$) \\
\vdash_{\mathcal{M}} & (\mathrm{¢}\, a(q'', req) b[ba]\$, \mathrm{¢}\, abbaabbp_0a\$) \\
\vdash_{\mathcal{M}} & (\mathrm{¢}\, a(q'', req) b[ba]\$, \mathrm{¢}\, abbaabb(p_0, res(a))a\$) \\
\vdash_{\mathcal{M}} & (\mathrm{¢}\, a(q'', rec(a)) b[ba]\$, \mathrm{¢}\, abbaabb(p_0, ack(a))a\$) \\
\vdash_{\mathcal{M}} & (\mathrm{¢}\, a(q_\$, req) b[ba]\$, \mathrm{¢}\, abbaabbap_0\$) \\
\vdash_{\mathcal{M}} & (\mathrm{¢}\, a(q_\$, req) b[ba]\$, \mathrm{¢}\, abbaabba(p_0, res(\$))\$) \\
\vdash_{\mathcal{M}} & (\mathrm{¢}\, a(q_\$, rec(\$)) b[ba]\$, \mathrm{¢}\, abbaabba(p_0, ack(\$))\$) \\
\vdash_{\mathcal{M}} & (\mathsf{Accept}, \mathsf{Accept}).
\end{array}
$$

Further, from the transition functions it is clear that if the input is not of the form of (w, ww), then at some point in the computation a communication fails. Finally, it is clear that M_1 and M_2 are deterministic and monotone.

3.2.2 On Deterministic and Monotone PC-Systems

Here we continue by comparing the input/output- and proper-relations of parallel communicating systems of det-mon-RRWW-automata with the relations derived in Subsection 3.1 for the single machines.

Proposition 3.2.2. *The classes* $\mathcal{R}\mathrm{el}_{io}$(det-mon-RRWW) *and* $\mathcal{R}\mathrm{el}_{io}$(det-mon-PC-RRWW) *are incomparable under inclusion.*

Proof. Clearly R_{copy} from Example 3.2.1 is not a pushdown relation for the reason that a regular language is mapped onto a context-sensitive language (see Section 2.3, "Properties"). Hence, as the relations computed by det-mon-RRWW-automata are properly contained in PDR, R_{copy} is not computable by any such automaton.

Conversely, the relation R_{pal} from Lemma 3.1.11 is computable by a det-mon-RRWW-automaton. But $R_{pal} \notin \mathcal{R}\mathrm{el}_{io}$(det-mon-PC-RRWW). Assume that $\mathcal{M} = (M_1, M_2)$ is a det-mon-PC-RRWW-system and $\mathrm{Rel}_{\mathrm{io}}(\mathcal{M}) = R_{\mathrm{pal}}$. Then in an accepting computation of $\mathcal{M}$, the component automaton M_2 just has tape contents $\mathrm{¢}c\$$. Thus, there are only finitely many different configurations that M_2 can reach in any accepting computation of $\mathcal{M}$.

Accordingly a non-forgetting RRWW-automaton M can be designed that simulates $\mathcal{M}$ as follows.

Using its tape M simulates M_1 step-by-step, while it simulates M_2 and all communication steps of $\mathcal{M}$ in its finite control. As M executes the exact cycles of M_1, it is monotone and deterministic, and it accepts the language $L(M) = \{ww^R \mid w \in \{a,b\}^*\}$. The class $\mathcal{L}$(det-mon-nf-RRWW) of languages accepted by non-forgetting monotone deterministic RRWW-automata coincides with the class of left-to-right regular languages (LRR) [MO11], which is a proper subclass of the class CRL of Church-Rosser languages. Thus, $L_{\text{pal}} = \{ww^R \mid w \in \{a,b\}^*\} \in \text{CRL}$ follows, contradicting the fact that L_{pal} is not a Church-Rosser language [JL02]. □

We continue with a technical and quite surprising result. It is well known that the non-forgetting property generally increases the power of restarting automata. In contrast we here show that this is not the case for parallel communicating systems of restarting automata.

Proposition 3.2.3. *For each deterministic non-forgetting* PC-RRWW*-system $\mathcal{M}$, there exists a deterministic* PC-RRWW*-system $\mathcal{M}'$ such that* $\text{Rel}_{\text{C}}(\mathcal{M}') = \text{Rel}_{\text{C}}(\mathcal{M})$. *In addition, if $\mathcal{M}$ is monotone, then so is $\mathcal{M}'$.*

Proof. Let $\mathcal{M} = (M_1, M_2)$ be a deterministic non-forgetting PC-RRWW-system. From $\mathcal{M}$ a deterministic PC-RRWW-system $\mathcal{M}' = (M_1', M_2')$ can be constructed such that M_1' and M_2' simulate M_1 and M_2, respectively, cycle by cycle. However, as M_1' and M_2' are reset to their respective initial state each time they execute a restart operation, they must determine the corresponding restart state of M_1 and M_2, respectively, by communicating with each other.

Here the main idea is that whenever M_1' is about to simulate a restart step of M_1, then it determines the restart state of M_1 and sends this information to M_2'. After having performed the corresponding restart step, M_1' requests the information about the correct restart state from M_2', and M_2' works similarly. There is, however, a serious problem with this approach. At the time when M_1' sends the information about the new restart state of M_1 to M_2', the automaton M_2' may already be waiting for a communication with M_1' that simulates a communication between M_2 and M_1. Then the communication newly initiated by M_1' will not correspond to the communication expected by M_2', and consequently the system $\mathcal{M}'$ may come to a deadlock. Thus, M_1' must make sure that M_2' has not yet entered

a communication before it attempts to send the information on the new restart state of M_1. Fortunately, these problems can be overcome by executing a two-way communication between M_1' and M_2' each time before a step of the computation of $\mathcal{M}$ is being simulated. This two-way communication is to ensure that both, M_1' and M_2', know the next step of both, M_1 and M_2, that they have to simulate.

We formalize this approach in the following. Again we want to simulate a det-nf-PC-RRWW-system $\mathcal{M} = (M_1, M_2)$, where

$$M_1 = (Q_1, \Sigma, \Gamma_1, ¢, \$, q_0, k, \delta_1) \text{ and } M_2 = (Q_2, \Delta, \Gamma_2, ¢, \$, p_0, k, \delta_2),$$

by a det-PC-RRWW-system $\mathcal{M}' = (M_1', M_2')$, where

$$M_1' = (Q_1', \Sigma, \Gamma_1, ¢, \$, q_0', k, \delta_1') \text{ and } M_2' = (Q_2', \Delta, \Gamma_2, ¢, \$, p_0', k, \delta_2').$$

As mentioned before the system $\mathcal{M}'$ must somehow save the information that the components of the non-forgetting system are able to carry from one cycle to the next. This will be achieved by additional communication steps between M_1' and M_2'. To this end we introduce the sets of *critical steps*

$$I_1 = \{rs(q) \mid q \in Q_1\} \cup \{acc, com, nc\}$$

and

$$I_2 = \{rs(p) \mid p \in Q_2\} \cup \{acc, com, nc\}$$

of M_1 and M_2, respectively. Here $rs(q)$ ($q \in Q_1$) denotes a restart step of M_1 in state q, *acc* denotes an accept step, *com* denotes a communication step, and *nc* denotes all other transition steps of M_1, and analogously for I_2 and M_2.

The basic idea underlying our simulation is the following. Each time that M_1' and M_2' want to simulate a computational step of M_1 and M_2, respectively, they first execute a "two-way handshake" in order to exchange information about the steps of M_1 and M_2 they are about to simulate. Accordingly, each step of M_1 and M_2 is simulated by a sequence of steps of M_1' and M_2' that looks as follows, where the exact definition of the symbols used will be explained below:

$$\begin{array}{lll} M_1': & \delta_1'(q,u) = (q,\mathrm{req}), \\ & \delta_1'((q,\mathrm{rec}(i)),u) = (q,\mathrm{res}(j)), \\ & \delta_1'((q,\mathrm{ack}(j)),u) = \delta_1(\dot{q},u); \\ M_2': & \delta_2'(p,v) = (p,\mathrm{res}(i)), \\ & \delta_2'((p,\mathrm{ack}(i)),v) = (p,\mathrm{req}), \\ & \delta_2'((p,\mathrm{rec}(j)),v) = \delta_2(\dot{p},v). \end{array} \tag{3.1}$$

Of course the last of these steps depends on i and j. According to our strategy i contains information on M_1 that M_2' sends to M_1', and symmetrically j contains information on M_2 that M_1' sends to M_2'. This information is taken from the sets I_1 and I_2, respectively. As M_1' and M_2' must store the corresponding information, we extend their sets of states by taking

$$Q_1' = \{[q,y] \mid q \in Q_1, y \in I_1 \cup I_2\} \text{ and } Q_2' = \{[p,x] \mid p \in Q_2, x \in I_1\}.$$

Further, we take $q_0' = [q_0, nc]$ and $p_0' = [p_0, nc]$.

Using these sets of states, the extended versions of the transition steps in (3.1) look as follows, where $x, x' \in I_1$, $y, y' \in I_2$, $q, \dot{q} \in Q_1$, $p, \dot{p} \in Q_2$, $u, u' \in (\Gamma_1 \cup \{¢, \$\})^*$, and $v, v' \in (\Gamma_2 \cup \{¢, \$\})^*$:

$$\begin{array}{lll} M_1': & \delta_1'([q,y],u) = ([q,y],\mathrm{req}), \\ & \delta_1'(([q,y],\mathrm{rec}(x,\langle nc,y'\rangle)),u) = ([q,x],\mathrm{res}(y,\langle x',y'\rangle)), \\ M_2': & \delta_2'([p,x],v) = ([p,x],\mathrm{res}(x,\langle nc,y'\rangle)); \\ & \delta_2'(([p,x],\mathrm{ack}(x,\langle nc,y'\rangle)),v) = ([p,x],\mathrm{req}). \end{array} \tag{3.2}$$

Here x contains the possibly lost information about the last step of M_1 that M_1' has just simulated, y denotes the last step of M_2 that M_2' has just simulated, x' is the next step of M_1 that M_1' is to simulate, and y' is the next step of M_2 that M_2' is to simulate. The latter are determined by M_1' and M_2' from the transition functions of M_1 and M_2, respectively, based on the current state q or p and on the current window contents u or v, respectively. Further observe that the two way "handshake" given in (3.2) is designed such that all needed information for the respective machine M_1' or M_2' is (here for M_1) encoded in the state $[q,x]$ and a triple of the form $(y,\langle x',y'\rangle)$. Finally this is done before the corresponding step of M_1 or M_2 is executed.

Now the exact form of the last transitions in (3.1) depends on the actual values of x, y, x', and y'. Here we present these transitions and corresponding simulations for the most

typical cases. To increase readability we use u and v (and some indexed versions of them) in several different contexts. However, as the following construction does not depend on the tape content it will be clear from the text in which context they are used.

1. The simplest case concerns the so-called non-critical (nc) steps. For M_1 these are the rewrite steps of the form $\delta_1(q,u) = (\dot{q}, u')$ and the move-right steps of the form $\delta_1(q,u) = (\dot{q}, \mathsf{MVR})$. As these steps do not depend in any way on M_2, M_1' transitions continue the steps given in (3.2) as follows for all possible values of y and y',

$$\begin{array}{rcll} \delta_1'(([q,nc], \mathrm{ack}(y, \langle nc, y'\rangle)), u) & = & ([\dot{q}, y'], u') & \text{or} \\ \delta_1'(([q,nc], \mathrm{ack}(y, \langle nc, y'\rangle)), u) & = & ([\dot{q}, y'], \mathsf{MVR}), & \end{array}$$

and analogously for M_2':

$$\begin{array}{rcll} \delta_2'(([p,x], \mathrm{rec}(nc, \langle x', nc\rangle)), v) & = & ([\dot{p}, x'], v') & \text{or} \\ \delta_2'(([p,x], \mathrm{rec}(nc, \langle x', nc\rangle)), v) & = & ([\dot{p}, x'], \mathsf{MVR}). & \end{array}$$

To illustrate these quite general steps we here present a simulation where both components of $\mathcal{M}$ perform a MVR-step, that is,

$$\mathcal{M}: \quad \cdots \vdash (\mathrm{¢}\, u_1 q u u_2 \$, \mathrm{¢}\, v_1 p v v_2 \$) \quad \vdash_{\mathsf{MVR}} \quad (\mathrm{¢}\, u_1 u \dot{q} u_2 \$, \mathrm{¢}\, v_1 v \dot{p} v_2 \$) \vdash \ldots \ .$$

Thus, $\mathcal{M}'$ mirrors these steps by the following computation:

$$\begin{array}{lll} \mathcal{M}': & \ldots & \\ & \vdash & (\mathrm{¢}\, u_1 [q,nc] u u_2 \$, \mathrm{¢}\, v_1 [p,nc] v v_2 \$) \\ & \vdash & (\mathrm{¢}\, u_1 ([q,nc], \mathrm{req}) u u_2 \$, \mathrm{¢}\, v_1 ([p,nc], \mathrm{res}(nc, \langle nc, nc\rangle)) v v_2 \$) \\ & \vdash & (\mathrm{¢}\, u_1 ([q,nc], \mathrm{rec}(nc, \langle nc, nc\rangle)) u u_2 \$, \mathrm{¢}\, v_1 ([p,nc], \mathrm{ack}(nc, \langle nc, nc\rangle)) v v_2 \$) \\ & \vdash & (\mathrm{¢}\, u_1 ([q,nc], \mathrm{res}(nc, \langle nc, nc\rangle)) u u_2 \$, \mathrm{¢}\, v_1 ([p,nc], \mathrm{req}) v v_2 \$) \\ & \vdash & (\mathrm{¢}\, u_1 ([q,nc], \mathrm{ack}(nc, \langle nc, nc\rangle)) u u_2 \$, \mathrm{¢}\, v_1 ([p,nc], \mathrm{rec}(nc, \langle nc, nc\rangle)) v v_2 \$) \\ & \vdash & (\mathrm{¢}\, u_1 u [\dot{q}, nc] u_2 \$, \mathrm{¢}\, v_1 v [\dot{p}, nc] v_2 \$) \\ & \ldots & . \end{array}$$

Observe that these non-critical steps can only be simulated directly, if the information x (y respectively) about the previous step that is given back during the two-way communication to both components M_1' and M_2' is non-critical. The following marks will concern exactly how the critical steps are simulated.

2. Next we consider the restart steps. Here we distinguish between the situation that only one of M_1 and M_2 is to execute a restart step, while the other component is in a non-restart configuration, and the situation that both components are to execute a restart step. The former case is easily solved. If M_1 is to execute the restart transition $\delta_1(q,u) = \mathsf{Restart}(\dot{q})$, then

$$\delta_1'(([q,nc],\mathrm{ack}(y,\langle rs(\dot{q}),y'\rangle)),u) \quad = \quad \mathsf{Restart}$$

for all $y \in \{rs(p), acc, com, nc\}$ $(rs(p) \in I_2)$, $y' \in \{acc, com, nc\}$, and if M_2 is to execute the restart transition $\delta_2(p,v) = \mathsf{Restart}(\dot{p})$, then

$$\delta_2'(([p,x],\mathrm{rec}(nc,\langle x',rs(\dot{p})\rangle)),v) \quad = \quad \mathsf{Restart}$$

for all $x' \in \{rs(q), acc, com, nc\}$ $(rs(q) \in I_2)$. Observe that the latter transition implies that the restart of M_2 does not depend on M_1. If, however, both, M_1 and M_2, are to execute a restart step, then we must ensure that M_1' and M_2' do not execute these restart steps simultaneously, as in this case the information about the corresponding restart states of M_1 and M_2 would get lost. Therefore, we give M_1' a "busy waiting round," during which it only communicates as shown in (3.2), that is, we take

$$\delta_1'(([q,nc],\mathrm{ack}(nc,\langle rs(\dot{q}),rs(\dot{p})\rangle)),u) \quad = \quad ([q,rs(\dot{p})],\mathrm{req}).$$

In other words, while M_1' "waits", M_2' is able to restart. Then in M_2''s next cycle M_1' sends its restart information and restarts.

After having performed a restart step, M_1' (or M_2') has no information on the corresponding restart state of M_1 (or M_2). Further, it does not know anything about the latest simulation step of M_2' (or M_1'), either. This information must be obtained from M_2' (or from M_1'). For M_1' this is easy. Immediately after having performed a restart step it receives the information $x = rs(\dot{q})$ as shown in (3.2). Accordingly it can continue as follows:

$$\delta_1'(([q_0,rs(\dot{q})],\mathrm{ack}(nc,\langle x',y'\rangle)),u) \quad = \quad ([\dot{q},y'],u),$$

where x' depends on $\delta_1'([\dot{q},y'],u)$.

For M_2' the situation is more complicated. During the "two-way-handshake" immediately after the restart operation, M_2' does not yet know the correct restart state of

M_2 when it sends the new value of y' to M_1' (see (3.2)). Hence, this value will in general be incorrect. M_2' can determine the correct value only after it has received the information about the correct restart state from M_1'. Therefore, M_2' has to execute an additional round of communication similar to the "busy waiting round" mentioned before, that is,

$$\delta_2'(([p_0, nc], \mathrm{rec}(rs(\dot{p}), \langle x', y' \rangle)), v) \quad = \quad ([\dot{p}, x'], \mathrm{res}(x', \langle nc, \hat{y} \rangle)),$$

where $\hat{y} \in I_2$ depends on $\delta_2'([\dot{p}, x'], v)$.

To illustrate the step that both component automata are to execute a restart operation we describe the corresponding simulation in detail. Thus, let $\mathcal{M}$ perform the following computation:

$$\begin{array}{lll} \mathcal{M}: & \ldots & \\ & \vdash & (\cent u_1 q u u_2 \$, \cent v_1 p v v_2 \$) \\ & \vdash_{\text{Restart of } M_1, M_2} & (\dot{q} \cent u_1 u u_2 \$, \dot{p} \cent v_1 v v_2 \$) \\ & \ldots . & \end{array}$$

According to its transition functions, $\mathcal{M}'$ behaves as described next:

$\ldots$

$\vdash \quad (\cent u_1 [q, nc] u u_2 \$, \cent v_1 [p, nc] v v_2 \$)$

$\vdash \quad (\cent u_1 ([q, nc], \mathrm{req}) u u_2 \$, \cent v_1 ([p, nc], \mathrm{res}(nc, \langle nc, rs(\dot{p}) \rangle)) v v_2 \$)$

$\vdash \quad (\cent u_1 ([q, nc], \mathrm{rec}(nc, \langle nc, rs(\dot{p}) \rangle)) u u_2 \$, \cent v_1 ([p, nc], \mathrm{ack}(nc, \langle nc, rs(\dot{p}) \rangle)) v v_2 \$)$

$\vdash \quad (\cent u_1 ([q, nc], \mathrm{res}(nc, \langle rs(\dot{q}), rs(\dot{p}) \rangle)) u u_2 \$, \cent v_1 ([p, nc], \mathrm{req}) v v_2 \$)$

$\vdash \quad (\cent u_1 ([q, nc], \mathrm{ack}(nc, \langle rs(\dot{q}), rs(\dot{p}) \rangle)) u u_2 \$, \cent v_1 ([p, nc], \mathrm{rec}(nc, \langle rs(\dot{q}), rs(\dot{p}) \rangle)) v v_2 \$)$

$\vdash \quad (\cent u_1 ([q, rs(\dot{p})], \mathrm{req}) u u_2 \$, [p_0, nc] \cent v_1 v v_2 \$)$ (Restart of M_2')

$\vdash \quad (\cent u_1 ([q, rs(\dot{p})], \mathrm{req}) u u_2 \$, ([p_0, nc], \mathrm{res}(nc, \langle nc, y' \rangle)) \cent v_1 v v_2 \$)$[19]

$\vdash \quad (\cent u_1 ([q, rs(\dot{p})], \mathrm{rec}(nc, \langle nc, y' \rangle)) u u_2 \$, ([p_0, nc], \mathrm{ack}(nc, \langle nc, y' \rangle)) \cent v_1 v v_2 \$)$

$\vdash \quad (\cent u_1 ([q, nc], \mathrm{res}(rs(\dot{p}), \langle rs(\dot{q}), y' \rangle)) u u_2 \$, ([p_0, nc], \mathrm{req}) \cent v_1 v v_2 \$)$

$\vdash \quad (\cent u_1 ([q, nc], \mathrm{ack}(rs(\dot{p}), \langle rs(\dot{q}), y' \rangle)) u u_2 \$, ([p_0, nc], \mathrm{rec}(rs(\dot{p}), \langle rs(\dot{q}), y' \rangle)) \cent v_1 v v_2 \$)$

$\vdash \quad ([q_0, nc] \cent u_1 u u_2 \$, ([\dot{p}, rs(\dot{q})], \mathrm{res}(rs(\dot{q}), \langle nc, y \rangle)) \cent v_1 v v_2 \$)$ (Restart of M_1')

[19] Here y' has not been determined correctly yet, and so M_2' has to take another round of communication.

$$
\begin{aligned}
&\vdash \quad (([q_0, nc], \mathrm{req}) \cent u_1 u u_2 \$, ([\dot{p}, rs(\dot{q})], \mathrm{res}(rs(\dot{q}), \langle nc, y\rangle)) \cent v_1 v v_2 \$) \\
&\vdash \quad (([q_0, nc], \mathrm{rec}(rs(\dot{q}), \langle nc, y\rangle)) \cent u_1 u u_2 \$, ([\dot{p}, rs(\dot{q})], \mathrm{ack}(rs(\dot{q}), \langle nc, y\rangle)) \cent v_1 v v_2 \$) \\
&\vdash \quad (([q_0, rs(\dot{q})], \mathrm{res}(nc, \langle x', y\rangle)) \cent u_1 u u_2 \$, ([\dot{p}, rs(\dot{q})], \mathrm{req}) \cent v_1 v v_2 \$) \\
&\vdash \quad (([q_0, rs(\dot{q})], \mathrm{ack}(nc, \langle x', y\rangle)) \cent u_1 u u_2 \$, ([\dot{p}, rs(\dot{q})], \mathrm{rec}(nc, \langle x', y\rangle)) \cent v_1 v v_2 \$) \\
&\vdash \quad ([\dot{q}, y] \cent u_1 u u_2 \$, [\dot{p}, x'] \cent v_1 v v_2 \$) \\
&\ldots \quad .
\end{aligned}
$$

3. Another critical situation occurs when M_1 or M_2 executes a communication step, as the simulation of this communication step must not interfere with the communication steps involved in the "two-way handshake." In this situation the component automaton that is to simulate this regular communication step announces this fact by choosing $x' = com$ or $y' = com$, and then it executes "busy waiting rounds" until the other component automaton reaches a regular communication step, too. Let us exemplarily show how the transition functions for M_1' and M_2' are derived if M_1 is to execute a regular request step $\delta_1(q, u) = (\dot{q}, \mathrm{req})$ and M_2 a response step $\delta_2(p, v) = (\dot{p}, \mathrm{res}(i))$.

   As mentioned before, both components of $\mathcal{M}'$ must enter the communication at the same time. Thus, if $y' \neq com$, M_1' takes a busy waiting by

$$\delta_1'(([q, nc], \mathrm{ack}(y, \langle com, y'\rangle)), u) \quad = \quad ([q, y'], \mathrm{req}),$$

   respectively for $x' \neq com$, M_2' also waits by

$$\delta_2'(([p, x], \mathrm{rec}(nc, \langle x', com\rangle)), v) \quad = \quad ([p, x'], \mathrm{res}(x', \langle nc, com\rangle)).$$

   Now both components announce that they are ready to communicate by the following two transitions:

$$
\begin{aligned}
\delta_1'(([q, nc], \mathrm{ack}(nc, \langle com, com\rangle)), u) &= ([\dot{q}, nc], \mathrm{req}), \\
\delta_2'(([p, nc], \mathrm{rec}(nc, \langle com, com\rangle)), v) &= ([\dot{p}, nc], \mathrm{res}(i)).
\end{aligned}
$$

   Obviously in terms of $\mathcal{M}$ also $\delta_1((\dot{q}, \mathrm{rec}(i)), u)$ and $\delta_2((\dot{p}, \mathrm{ack}(i)), v)$ must be defined and can be treated like every other step before, that is, after entering the regular communication $\mathcal{M}'$ continues the two-way handshake in equation (3.2) with

$$\delta_1'(([\dot{q}, nc], \mathrm{rec}(i)], u) = ([\dot{q}, nc], \mathrm{req})$$

and

$$\delta_2'(([\dot{p}, nc], \mathrm{ack}(i)), v) = ([\dot{p}, nc], \mathrm{res}(nc, \langle nc, y' \rangle)).$$

In this case we describe an exemplary computation of $\mathcal{M}$, where M_2 waits in a response state until M_1 enters the corresponding communication state. Here c and $\hat{c}$ denote arbitrary (no communication) configurations of M_1;

$$\begin{array}{lll}
\mathcal{M}: & \ldots & \\
& \vdash & (c, \not\in v_1 p v v_2 \$) \\
& \vdash & (\hat{c}, \not\in v_1 (\dot{p}, \mathrm{res}(i)) v v_2 \$) \\
& \vdots & \\
& \vdash & (\not\in u_1 q u u_2 \$, \not\in v_1 (\dot{p}, \mathrm{res}(i)) v v_2 \$) \\
& \vdash & (\not\in u_1 (\dot{q}, \mathrm{req}) u u_2 \$, \not\in v_1 (\dot{p}, \mathrm{res}(i)) v v_2 \$) \\
& \vdash & (\not\in u_1 (\dot{q}, \mathrm{rec}(i)) u u_2 \$, \not\in v_1 (\dot{p}, \mathrm{ack}(i)) v v_2 \$) \\
& \ldots . &
\end{array}$$

$\mathcal{M}'$ mirrors the above computation by a sequence of transition steps of the following form, where $c', \hat{c}'$ are the corresponding configurations to $c, \hat{c}$ of M_1' and $x, x' \neq com$:

$$\begin{array}{lll}
\mathcal{M}': & \ldots & \\
& \vdash & (c', \not\in v_1 [p, x] v v_2 \$) \\
& \vdash & (\ldots, \not\in v_1 ([p, x], \mathrm{res}(x, \langle nc, com \rangle)) v v_2 \$) \\
& \vdots & {}^{20} \\
& \vdash & (\hat{c}', \not\in v_1 [p, x'] v v_2 \$) \\
& \vdash & (\ldots, (\not\in v_1 [p, x'], \mathrm{res}(x', \langle nc, com \rangle)) v v_2 \$) \\
& \vdots & {}^{20} \\
& \vdash & (\not\in u_1 ([q, nc], \mathrm{ack}(nc, \langle com, com \rangle)) u u_2 \$, \not\in v_1 ([p, nc], \mathrm{rec}(nc, \langle com, com \rangle)) v v_2 \$) \\
& \vdash & (\not\in u_1 ([\dot{q}, nc], \mathrm{req}) u u_2 \$, \not\in v_1 ([\dot{p}, nc], \mathrm{res}(i)) v v_2 \$) \\
& \vdash & (\not\in u_1 ([\dot{q}, nc], \mathrm{rec}(i)) u u_2 \$, \not\in v_1 ([\dot{p}, nc], \mathrm{ack}(i)) v v_2 \$) \\
& \vdash & (\not\in u_1 ([\dot{q}, nc], \mathrm{req}) u u_2 \$, \not\in v_1 ([\dot{p}, nc], \mathrm{res}(nc, \langle nc, y' \rangle))) v v_2 \$) \\
& \vdash & \ldots
\end{array}$$

[20] Here M_2' waits by performing continually the two-way handshake until M_1' reaches *com*, either.

We omit to describe the other communication direction, as it is similar to the one shown above.

4. When one of M_1 or M_2 executes an accept step, then the corresponding component automaton M_1' or M_2' cannot simply accept, as that might again interfere with the "two-way handshakes." Accordingly, the component automaton that is to simulate this accept step announces this fact by choosing $x' = acc$ or $y' = acc$, and then it executes "busy waiting rounds" until the other component automaton reaches an accept step, too. The transition function in that case is defined exactly the same as shown in mark 3. Hence, we omit further details here.

This covers all major cases. Now based on the above outline the transition functions δ_1' and δ_2' for all remaining cases can be easily defined, and by the composition of the simulation parts above it is clear that $\mathcal{M}'$ does indeed simulate $\mathcal{M}$ step by step. In particular, it follows that $\mathrm{Rel_C}(\mathcal{M}') = \mathrm{Rel_C}(\mathcal{M})$, and as the simulation is in some sense quite direct it also follows that $\mathcal{M}'$ is monotone, if $\mathcal{M}$ is. □

Observe again that the previous proof is a quite direct simulation of a non-forgetting PC-system by a forgetting one and that it does not depend in any way on the tape content. Therefore, we strongly suspect that the proof also works for any type of (non-deterministic) PC-system of restarting automata. Anyway, we here omit further investigations, as the next result already shows that every computable relation can be characterized by the proper relations of two deterministic and monotone devices.

Theorem 3.2.4. *Let $R \subseteq \Sigma^* \times \Delta^*$ be a relation. Then $R \in \mathcal{R}\mathrm{el_P}(\mathsf{det\text{-}mon\text{-}PC\text{-}RRWW})$ if and only if it is computable.*

Proof. Certainly a PC-RRWW-system can be simulated by a Turing machine. Thus, given a pair $(u, v) \in \mathrm{Rel_P}(\mathcal{M})$, where $\mathcal{M}$ is a monotone deterministic PC-RRWW-system, a Turing Machine T can non-deterministically guess words $x \in \Gamma_1^*$ and $y \in \Gamma_2^*$ satisfying $\mathsf{Pr}^{\Sigma}(x) = u$ and $\mathsf{Pr}^{\Delta}(y) = v$, and then it can simulate $\mathcal{M}$ starting from the initial configuration $K_{\mathrm{in}}(x, y)$. Thus, the transduction $\mathrm{Rel_P}(\mathcal{M})$ is computable.

Conversely, let $R \subseteq \Sigma^* \times \Delta^*$ be a relation that is computable. Thus, there exists a Turing Machine T for R. Actually there are several ways to associate a relation to unrestricted Turing Machines that obviously all coincide. Here $T = (Q, \Sigma, \Gamma, \delta, q_0^{(T)}, \square, F)$

is a non-deterministic device, where $\Delta \subseteq \Gamma$, that, given $u \in \Sigma^*$ as input, has an accepting computation that ends with the result $v \in \Delta^*$ on the tape if and only if the pair (u, v) belongs to R. For our purposes we need to convert T to a Turing Machine $T' = (Q', \bar{\Sigma}, \Gamma', \delta', q_0^{(T')}, \Box, q_F)$ as follows:

- The input alphabet Σ is replaced by a new alphabet $\bar{\Sigma} = \{\bar{a} \mid a \in \Sigma\}$, that is a marked copy of Σ.
- The output alphabet Δ is also replaced by a new alphabet $\bar{\Delta} = \{\bar{c} \mid c \in \Delta\}$.
- The original alphabets Σ and Γ are disjoint from the new tape alphabet Γ'.
- Instead of having a set of final states F, q_F is the only final state for T'.
- Each accepting computation of T' consists of an odd number of steps.

Clearly, there is a Turing Machine T' for every Turing Machine T, such that, given $\bar{u} \in \bar{\Sigma}^*$ (that is the encoded version of $u \in \Sigma^*$) as input, it accepts with the result encoded as $\bar{v} \in \bar{\Delta}^*$ as output.

From this Turing machine we now construct a monotone deterministic PC-RRWW-system $\mathcal{M} = (M_1, M_2)$ such that $\mathrm{Rel_P}(\mathcal{M}) = R$, where $M_1 = (Q_1, \Sigma, \Gamma_1, \mathfrak{c}, \$, q_0, k_1, \delta_1)$ and $M_2 = (Q_2, \Delta, \Gamma_2, \mathfrak{c}, \$, p_0, k_2, \delta_2)$. Because of Proposition 3.2.3 we can describe $\mathcal{M}$ as a non-forgetting PC-RRWW-system. Let $\Gamma_1 = \Sigma \cup Q' \cup \Gamma' \cup \{\#\}$ and $\Gamma_2 = \Delta \cup Q' \cup \Gamma' \cup \{\#\}$ be the tape alphabets of M_1 and M_2, respectively.

We define the transition functions δ_1 and δ_2 such that the characteristic transduction $\mathrm{Rel_C}(\mathcal{M})$ of $\mathcal{M}$ will consist of all pairs of words $(x, y) \in \Gamma_1^* \times \Gamma_2^*$ that mirror a valid computation of T'. Hence, for (x, y) the following conditions must be satisfied:

For $u \in \Sigma^*$ and $v \in \Delta^*$ there is an accepting computation of T' of the form

$$q_0^{(T')}\bar{u} \vdash_{T'} x_1q_1y_1 \vdash_{T'} \cdots \vdash_{T'} x_{n-2}q_{n-2}y_{n-2} \vdash_{T'} x_{n-1}q_{n-1}y_{n-1} \vdash_{T'} q_F\bar{v},$$

if and only if

$$\begin{array}{lrcl} (i) & x & = & \#u\#\#x_1q_1y_1\#\#x_3q_3y_3\#\#\ldots\#\#x_{n-2}q_{n-2}y_{n-2}\#\#q_F\bar{v}, \text{ and} \\ (ii) & y & = & \#\#q_0^{(T')}\bar{u}\#\#x_2q_2y_2\#\#\ldots\#\#x_{n-1}q_{n-1}y_{n-1}\#v. \end{array}$$

Next we describe the behavior of the non-forgetting restarting automata M_1 and M_2:

1. Starting from its initial state M_1 expects to have a tape contents x from the regular set $E_1 = \# \cdot \Sigma^* \cdot (\#\# \cdot \Gamma'^* \cdot Q' \cdot \Gamma'^*)^* \cdot \#\# \cdot q_F \cdot \bar{\Delta}^*$, and M_2 expects to have a tape contents y from the regular set $E_2 = \#\# \cdot q_0^{(T')} \cdot \bar{\Sigma}^* \cdot (\#\# \cdot \Gamma'^* \cdot Q' \cdot \Gamma'^*)^* \cdot \# \cdot \Delta^*$. During its first cycle M_1 erases the first occurrence of the symbol # from its tape, it checks that the factor $u \in \Sigma^*$ on its tape corresponds to the factor from $\bar{\Sigma}^*$ on M_2's tape using communications, and it verifies that its tape contents belongs to the regular language E_1. Analogously, M_2 also erases the first occurrence of the symbol # from its tape, and it verifies that its tape contents belongs to the regular set E_2. If all these tests are successful, then both M_1 and M_2 restart in particular non-initial states; otherwise, the computation fails.

2. In the next cycle M_1 and M_2 check by communication that the first factor marked by ## on M_1's tape is an immediate successor configuration of the factor marked by a single symbol # on M_2's tape with respect to the computation relation of the Turing Machine T'. During this process each of M_1 and M_2 erases the leftmost occurrence of the symbol # from its tape. In the affirmative, both M_1 and M_2 restart in non-initial states; otherwise, the computation fails.

3. In the next cycle the roles of M_1 and M_2 are interchanged.

4. The last two steps are repeated until the syllable $q_F\bar{v}$ on M_1's tape is reached. In this case the words x and y do indeed describe an accepting computation of T_2 that produces the result $\bar{v}$ starting from $\bar{u}$. Now in a tail computation M_1 and M_2 compare the factor $\bar{v}$ on M_1's tape to the suffix $v \in \Delta^*$ on M_2's tape by communications. If $\bar{v} = v$, then both M_1 and M_2 accept, as in this case x and y satisfy all the conditions stated above; otherwise, the computation fails.

Based on the description above it is obvious how δ_1 and δ_2 have to be defined. It follows from this description that $\mathrm{Rel_C}(\mathcal{M})$ is indeed the transduction defined above. Hence, the projection of $\mathrm{Rel}_C(\mathcal{M})$ onto Σ and Δ yields that $\mathrm{Rel_P}(\mathcal{M}) = R$ holds. Further, observe that M_1 and M_2 are both monotone and deterministic, which completes the proof. □

Together with Proposition 3.2.2 this result yields the following proper inclusions.

Corollary 3.2.5.

$$\begin{array}{lrcl} \text{(a)} & \mathcal{R}\mathrm{el}_P(\textsf{det-mon-RRWW}) & \subsetneq & \mathcal{R}\mathrm{el}_P(\textsf{det-mon-PC-RRWW}). \\ \text{(b)} & \mathcal{R}\mathrm{el}_{io}(\textsf{det-mon-PC-RRWW}) & \subsetneq & \mathcal{R}\mathrm{el}_P(\textsf{det-mon-PC-RRWW}). \end{array}$$

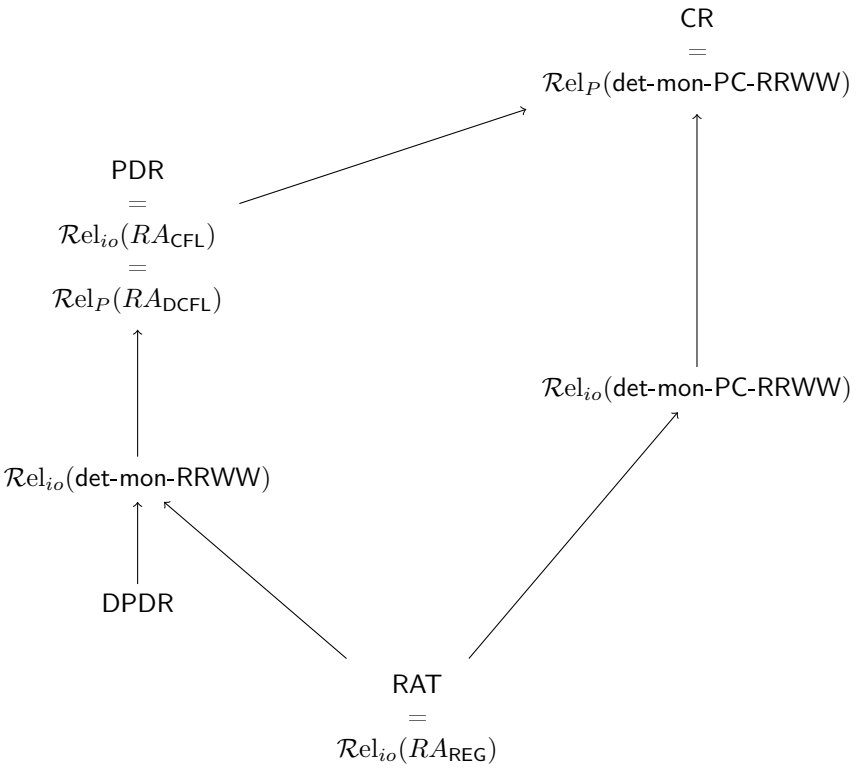

Figure 3.1: Taxonomy of classes of relations computed by types of monotone deterministic restarting automata. Here RA_{REG}, RA_{DCFL} and RA_{CFL} denote the sets of restarting automata that characterize the regular, deterministic context-free and context-free languages, respectively.

We summarize the relationships between the various classes of transductions considered in this chapter by the diagram in Figure 3.1, where an arrow denotes a proper inclusion, and classes that are not connected are incomparable under inclusion.

Chapter 4

Restarting Transducers

According to Berstel ([Ber79], p.53) relations and transductions simply offer two different perspectives of the same object. In his sense relations (i.e. sets) provide a more "static" point of view, while transductions (i.e. mappings) follow the "dynamic" aspect, implied by pairing words. Although we know that, from a set theoretic point of view, both concepts are equivalent, there is a difference when we turn to machines that realize relations or transductions. The present work illustrates these different perspectives. While in the previous chapter we associated relations with restarting automata such that they expect pairs of words as input, here we change our perspective to the dynamic aspect of relations. In the spirit of traditional transducing devices (see Section 2.3) we consider a new model of transducer, based on restarting automata. We call these machines "restarting transducers". A restarting transducer is a restarting automaton with explicit output (see Figure 4.1).

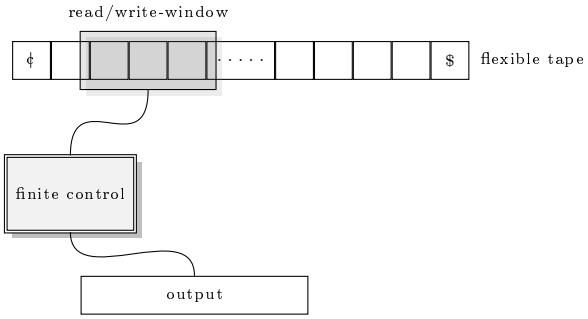

Figure 4.1: Schematic representation of a restarting transducer.

4.1 Definition, Examples, and General Observations

Here we extend the definition of restarting automata to transducers. For that we mainly refer to the formal definitions given for restarting automata in Section 2.2. As, however, most of the presented formalisms carry over to restarting transducers, we shorten some of the already known formal aspects in the following.

Concerning the different types of restarting automata we take the RRWW-automaton to be the basis of the formal definition of the respective transducer. Thus, a *restarting transducer* (RRWW-Td for short) is described as a 9-tuple $T = (Q, \Sigma, \Delta, \Gamma, ¢, \$, q_0, k, \delta)$. Again we have Q as the finite set of states, Σ and Γ as the finite input and tape alphabet ($\Sigma \subseteq \Gamma$), $¢, \$ \notin \Gamma$ as the markers for the left and right border of the tape, $q_0 \in Q$ as the initial state and $k \geq 1$ as the size of the read/write window. Additionally the description contains the finite output alphabet Δ, and the transition function δ extends to

$$\delta : Q \times \mathcal{PC}^{(k)} \to \mathfrak{P}_{fin}(Q \times (\{\mathsf{MVR}\} \cup \mathcal{PC}^{\leq (k-1)}) \cup \{\mathsf{Restart}, \mathsf{Accept}\} \times \Delta^*).$$

Based on the definition of δ, it is obvious that the transducer, just like the automaton, is in general non-deterministic. It works in cycles, where each cycle is a combination of a number of *move-right* steps, one *rewrite* step, and a *restart* or *accept* step[21]. Output letters, that is, a word from Δ^*, are produced at the end of every cycle (during a restart step) and during an accept step. Actually the latter causes the only difference in the formal definition of transducers. Thus, the restart and accept steps are now of the form $(\mathsf{Restart}, y) \in \delta(q, x)$ ($(\mathsf{Accept}, y) \in \delta(q, x)$, respectively), where $q \in Q$, $x \in \mathcal{PC}^{(k)}$ and $y \in \Delta^*$ is the output assigned to the restart or accept step. In the following we give a detailed explanation of these significant steps. For that we first introduce the notion of configurations.

A *configuration* of a restarting transducer $T = (Q, \Sigma, \Delta, \Gamma, ¢, \$, q_0, k, \delta)$ is described as a pair $(\alpha q \beta, z)$, where $\alpha q \beta$ ($\alpha\beta \in ¢ \cdot \Gamma^* \cdot \$$, $q \in Q$) is the configuration of the underlying restarting automaton[22] and $z \in \Delta^*$ is the output produced so far. Initially T is in the configuration $(q_0 ¢ u \$, \varepsilon)$ (called the *initial configuration*,), where $u \in \Sigma^*$ is the input word and the output is empty. The *restarting configuration* is described by $(q_0 ¢ u' \$, v)$, where $u' \in \Gamma^*$ and $v \in \Delta^*$ is the current output. Finally (Accept, z) denotes an accepting configuration, where $z \in \Delta^*$ is the output word of T.

[21] In case of a tail computation the rewrite step is optional.

[22] Observe that the underlying restarting automaton can easily be obtained from the transducer.

Accordingly, an *accepting computation* of T on input $u \in \Sigma^*$ consists of a finite sequence of cycles that is followed by an accepting tail computation. It can be described as

$$(q_0 \mathbin{\text{¢}} u\$, \varepsilon) \vdash_T^c (q_0 \mathbin{\text{¢}} u_1\$, v_1) \vdash_T^c \ldots \vdash_T^c (q_0 \mathbin{\text{¢}} u_n\$, v_1 \cdots v_n) \vdash_T^* (\mathsf{Accept}, v_1 \cdots v_n v_{n+1}),$$

where $u_1, ..., u_n \in \Gamma^*$ and $v_1, ..., v_{n+1} \in \Delta^*$. Obviously $\vdash_T$ denotes the extended[23] single step relation and $\vdash_T^c$ denotes the execution of a complete cycle, known from restarting automata. Finally, $\vdash_T^*$ and $\vdash_T^{c^*}$ are the reflexive and transitive closures of these relations. Note that within a cycle the behavior of the restarting transducer coincides with the behavior of the corresponding restarting automaton, hence, we here omit further details.

Now the restarting transducer $T = (Q, \Sigma, \Delta, \Gamma, \text{¢}, \$, q_0, k, \delta)$ realizes the *transduction* $T : \Sigma^* \to \Delta^*$ that is defined as follows, for every $u \in \Sigma^*$:

$$T(u) = \{v \in \Delta^* \mid (q_0 \mathbin{\text{¢}} u\$, \varepsilon) \vdash_T^* (\mathsf{Accept}, v)\}.$$

Clearly the image of a language $L \subseteq \Sigma^*$ under T is defined as $T(L) = \bigcup_{u \in L} T(u)$ and the preimage of a language $L' \subseteq \Delta^*$ is $T^{-1}(L') = \{u \in \Sigma^* \mid T(u) \cap L' \neq \emptyset\}$. From a static point of view, the *relation* computed by T is the graph of its transduction, that is,

$$\text{Rel}(T) = \{(u, v) \mid v \in T(u)\}.$$

Finally the *class of relations* defined by a type of restarting transducer (here RRWW-Td) is denoted by $\mathcal{R}$el(RRWW-Td).

Before we come to a first example we should recall the notion of *meta-instructions*. Meta-instructions were used to increase the readability of the behavior of restarting automata (see Section 2.2, "Basic Properties"). Hence, a tuple of the form $(E_1, u \mapsto u', E_2)$ mirrors one cycle of an RRWW-automaton, where it reads across the tape content E_1, rewrites a sub-word u by the shorter sub-word u' and finally verifies that the remaining suffix on the tape corresponds to E_2. As these meta-instructions describe the rewriting behavior of an automaton, they can easily be extended to restarting transducers. Here

$$(E_1, u \mapsto u', E_2; v)$$

is a meta-instruction of a restarting transducer, where E_1, E_2, u, u' are defined as for the corresponding automaton, and v is the output word produced at the end of this cycle.

[23] The relation is now defined on tuples to additionally mirror the behavior of the output function.

Accordingly, $(\mathfrak{c} \cdot E \cdot \$, \mathsf{Accept}; v)$ denotes an accepting meta-instruction.

Example 4.1.1. Let $T = (Q, \{a,b,c\}, \{\hat{a},\hat{b},\hat{c}\}, \{a,b,c,B,C\}, \mathfrak{c}, \$, q_0, 3, \delta)$ be the RRWW-Td that is described by the following meta-instructions:

$$\begin{array}{llllll}
(1) & (\mathfrak{c}\cdot(abc)^*, & abc \to Bc, & (Bc)^*\cdot\$; & \hat{a}),\\
(2) & (\mathfrak{c}\cdot(Bc)^*, & Bc \to C, & C^*\cdot\$; & \hat{b}),\\
(3) & (\mathfrak{c}\cdot C^*, & C \to \varepsilon, & \$; & \hat{c}),\\
(4) & (\mathfrak{c}\cdot\$, & \mathsf{Accept}; & & \varepsilon).
\end{array}$$

Obviously T consumes only words from the input language $L = \{(abc)^n | n \geq 0\}$. Thus, at the beginning of the computation it scans the tape from left to right while checking the correct order of a's, b's and c's. Next T deletes stepwise all a's from right to left and produces the same number of $\hat{a}$'s. Then, the transducer proceeds to do the same for b's and c's. In order to do so it is clear that $T(L) = \{\hat{a}^n\hat{b}^n\hat{c}^n \mid n \geq 0\}$ and the graph of that transduction is $\mathrm{Rel}(T) = \{((abc)^n, \hat{a}^n\hat{b}^n\hat{c}^n) | n \geq 0\}$. From the meta-instructions it is clear how the transition function can be designed. Implicitly we describe δ by exemplarily showing how T proceeds on input *abcabcabc*:

$$\begin{array}{lll|lll}
(q_0\mathfrak{c}abcabcabc\$, \varepsilon) & \vdash_{\mathsf{MVR}} & (\mathfrak{c}q_1abcabcabc\$, \varepsilon) & \vdash_{\mathsf{MVR}} & (\mathfrak{c}aq_2bcabcabc\$, \varepsilon)\\
& \vdash_{\mathsf{MVR}} & (\mathfrak{c}abq_3cabcabc\$, \varepsilon) & \vdash_{\mathsf{MVR}} & (\mathfrak{c}abcq_1abcabc\$, \varepsilon)\\
& \vdash^*_{\mathsf{MVR}} & (\mathfrak{c}abcabcq_1abc\$, \varepsilon) & \vdash_{\mathsf{Rewrite}} & (\mathfrak{c}abcabcBcq_1'\$, \varepsilon)\\
& \vdash_{\mathsf{Restart}} & (q_0\mathfrak{c}abcabcBc\$, \hat{a}) & \vdash^*_{\mathsf{MVR}} & (\mathfrak{c}abcq_1abcBc\$, \hat{a})\\
& \vdash_{\mathsf{Rewrite}} & (\mathfrak{c}abcBcq_1'Bc\$, \hat{a}) & \vdash_{\mathsf{Restart}} & (q_0\mathfrak{c}abcBcBc\$, \hat{a}\hat{a})\\
& \vdash_{\mathsf{MVR}} & (\mathfrak{c}q_1abcBcBc\$, \hat{a}\hat{a}) & \vdash_{\mathsf{Rewrite}} & (\mathfrak{c}Bcq_1'BcBc\$, \hat{a}\hat{a})\\
& \vdash_{\mathsf{MVR}} & (\mathfrak{c}BcBq_2'cBc\$, \hat{a}\hat{a}) & \vdash_{\mathsf{MVR}} & (\mathfrak{c}BcBcq_1'Bc\$, \hat{a}\hat{a})\\
& \vdash_{\mathsf{Restart}} & (q_0\mathfrak{c}BcBcBc\$, \hat{a}\hat{a}\hat{a}) & &\\
& \vdash^c & (q_0\mathfrak{c}BcBcC\$, \hat{a}\hat{a}\hat{a}\hat{b}) & \vdash^c & (q_0\mathfrak{c}BcCC\$, \hat{a}\hat{a}\hat{a}\hat{b}\hat{b})\\
& \vdash^c & (q_0\mathfrak{c}CCC\$, \hat{a}\hat{a}\hat{a}\hat{b}\hat{b}\hat{b}) & \vdash^{c^*} & (q_0\mathfrak{c}\$, \hat{a}\hat{a}\hat{a}\hat{b}\hat{b}\hat{b}\hat{c}\hat{c}\hat{c})\\
& \vdash_{\mathsf{Accept}} & (\mathsf{Accept}, \hat{a}\hat{a}\hat{a}\hat{b}\hat{b}\hat{b}\hat{c}\hat{c}\hat{c}). & &
\end{array}$$

Observe that T is non-deterministic, that is, it guesses the correct position of every rewrite step.

Modes of Operation

All the modifications presented in Section 2.2 (see "Variants of Restarting Automata") for restarting automata naturally carry over to transducers. Hence, we distinguish between R, RR, RW, RRW, RWW, RRWW, *deterministic* (det-), *monotone* (mon-) and *non-forgetting* (nf-) restarting transducers. Note that in contrast to the situation for pushdown transducers (see Section 2.3, "Pushdown Relations") the relation computed by any deterministic restarting transducer is actually a (partial) function.

We here introduce a further property, unique for restarting transducers and useful for the following reflections. A restarting transducer is called *proper* (prop- for short) if all its accept instructions are of the form $(\mathsf{Accept}, \varepsilon)$, that is, in the last step of an accepting computation it can only output the empty word.

Concerning all the modifications introduced above, it will be one of the major topics of the present work to investigate the effect of these different mechanisms on the computational power of restarting transducers.

4.1.1 General Observations

Before we turn to more restricted versions of restarting transducers we will establish some general observations. The first ones are the corresponding versions of the repeatedly used error and correctness preserving properties (cf. Propositions 2.2.3 and 2.2.4) for restarting automata.

Proposition 4.1.2 (Error Preserving Property for Restarting Transducers). *Let* $T = (Q, \Sigma, \Delta, \Gamma, \mathfrak{c}, \$, q_0, k, \delta)$ *be an* RRWW*-transducer, and let* (u, v) *and* (u', v') *be pairs of words over* $\Sigma^* \times \Delta^*$. *If* $(q_0 \mathfrak{c} u \$, v) \vdash_T^{c^*} (q_0 \mathfrak{c} u' \$, v')$ *holds and* $(u, v) \notin \mathrm{Rel}(T)$, *then* $(u', v') \notin \mathrm{Rel}(T)$, *either.*

Proposition 4.1.3 (Correctness Preserving Property for Restarting Transducers). *Let* $T = (Q, \Sigma, \Delta, \Gamma, \mathfrak{c}, \$, q_0, k, \delta)$ *be an* RRWW*-transducer, and let* (u, v) *and* (u', v') *be pairs of words over* $\Sigma^* \times \Delta^*$. *If* $(q_0 \mathfrak{c} u \$, v) \vdash_T^{c^*} (q_0 \mathfrak{c} u' \$, v')$ *is an initial segment of an accepting computation of* T, *then* $(u', v') \in \mathrm{Rel}(T)$.

A property introduced for relations in general (see Definition 2.1.2) helps us to achieve a first classification of restarting transductions. Thus, the following result is based on the

fact that a restarting transducer is not able to perform arbitrary many restart steps[24].

Proposition 4.1.4. *Every relation computed by a restarting transducer is length bounded.*

Observe that the property of being length-bounded does not depend on the length of the output produced in a single step. In fact, it only depends on whether a transducing system is able to produce output symbols without consuming any input symbols.

An immediate consequence of this observation is the next corollary.

Corollary 4.1.5. *There is no restarting transducer that computes the relation* $R = \{(\varepsilon, c^n) \mid n \geq 0\}$.

Not surprisingly, Proposition 4.1.4 will lead in the following to some important incomparability results, as most of the classical types of transducers are not bounded in the use of ε-steps in their computations. Anyway, here we continue with a general classification of some subclasses of RRWW-transducers.

Proposition 4.1.6. *Let X_2 be any type of restarting automaton, and let X_1 be a type of restarting automaton that is a restricted version of X_2*[25]*. Now let X_1-Td, X_2-Td be the corresponding transducer classes. If $\mathcal{L}(X_1) \subsetneq \mathcal{L}(X_2)$, then $\mathcal{R}\text{el}(X_1\text{-Td}) \subsetneq \mathcal{R}\text{el}(X_2\text{-Td})$.*

Proof. The proof of Proposition 4.1.6 is quite obvious. Nevertheless it is worth to give a deeper inside as it is a good example to restate some basic facts on the connection between languages and relations.

By definition it is clear that one relation class includes the other. This inclusion is proper for the following reason. Let L be a language that can be computed by $M_2 \in X_2$ and there exists no restarting automaton $M_1 \in X_1$ that is able to do so. Hence, $L \in \mathcal{L}(X_2)$ and $L \notin \mathcal{L}(X_1)$. The semi-characteristic function of L is defined as

$$\chi'_L(w) = \begin{cases} 1 & ; w \in L \\ undefined & ; w \notin L \end{cases}.$$

Obviously a restarting automaton for L can easily be extended to a restarting transducer that computes χ'_L by writing the symbol 1 on the output tape during an accept step.

[24] In fact the number of restart steps is bounded by the length of the input, as at least one symbol has to be deleted in any cycle.

[25] For instance, an RW-automaton is a restricted version of an RWW-automaton.

It follows that there exists a restarting transducer of type X_2-Td that computes χ'_L and further on, there is no restarting transducer of type X_1-Td for χ'_L. Notice that the same technique does not work for computing the characteristic function

$$\chi_L(w) = \begin{cases} 1 & ; w \in L \\ 0 & ; w \notin L \end{cases}$$

of any language L. For that, let L be accepted by a non-deterministic restarting automaton. Hence, for a word $w \in L$, there might exist several paths in the computation tree, some accepting, some not. Observe that when we convert this automaton into a transducer for the characteristic function, then all paths must be accepting, some with 0 and some with 1 as output. Now, as considered above, we can clearly assign the corresponding output to the accept steps or reject steps. But then, obviously, the transducer produces on input w (from above) two different outputs. Hence, it computes a relation that is not a function.[26] □

For the same reason the following result holds.

Corollary 4.1.7. *Let X_1, X_2 be two different classes of restarting automata. If $\mathcal{L}(X_1)$ and $\mathcal{L}(X_2)$ are incomparable, so are $\mathcal{R}\text{el}(X_1\text{-Td})$ and $\mathcal{R}\text{el}(X_2\text{-Td})$.*

Note that the converse direction of the previous statement does not hold in general. In summary these observations and the results for the different language classes of restarting automata (see Figure 2.2) lead to the inclusion diagram given in Figure 4.2. For that, recall that deterministic types of automata only compute functions, while non-deterministic ones compute "real" relations. Therefore, a class of restarting transducers of type det-X is naturally a proper subclass of the non-deterministic version X. Further, if two types of restarting automata are equivalent, then the classes of relations computed by the corresponding transducers need not coincide. Hence, it is not clear whether the equivalence $\mathcal{L}(\text{det-RWW}) = \mathcal{L}(\text{det-RRWW})$ carries over to relations or not. In addition the open questions whether the inclusion $\mathcal{R}\text{el}(\text{RWW-Td}) \subseteq \mathcal{R}\text{el}(\text{RRWW-Td})$ is proper and whether $\mathcal{R}\text{el}(\text{RRW-Td})$ is contained in $\mathcal{R}\text{el}(\text{RWW-Td})$ are variants of the corresponding questions for restarting automata (see Section 2.2, "General Classifications"). Discussions on these question can be found in Chapters 5 and 6.

[26] Indeed the word problem is decidable for any type of restarting automaton, but this does not imply that the characteristic function of a restarting automaton of a certain type is computable by a restarting transducer of the corresponding type.

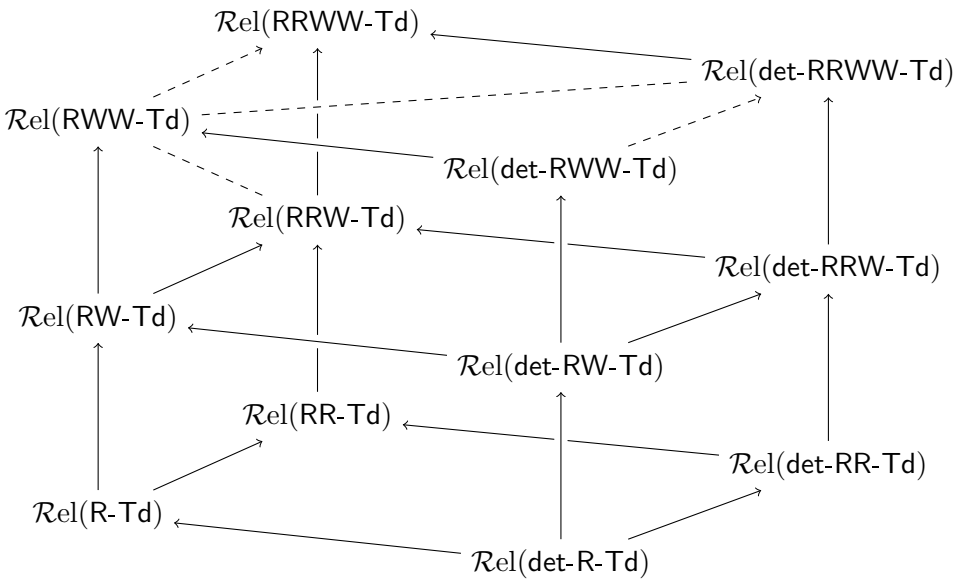

Figure 4.2: Inclusions between the relation classes defined by the basic types of restarting transducers. Proper inclusions are denoted by arrows, inclusions not known to be proper by dashed arrows, and unknown relationships by dashed lines.

An upper bound ($\mathsf{NP} \cap \mathsf{CSL}$) for the power of restarting automata was given in Proposition 2.2.7. One might say that this class, in some sense, also serves as an upper bound for the corresponding transducer classes. However, here we present a different criterion, unique for transducers, that serves as an indicator for the computational power of restarting transducers. We start our observations by focusing on the output language classes of these machines. For that the following definition as well as a result on det-R-automata from [NO99] (see p.10) is needed.

Definition 4.1.8. *A language class $\mathcal{C}$ is called a quotient basis for the recursively enumerable languages if, for each language $L \in \mathsf{RE}$, there exists a language $L' \in \mathcal{C}$ and a regular set R such that $L = R\backslash L'$. Here $R\backslash L'$ denotes the left-quotient of L' by R, that is, $R\backslash L' = \{w \mid \exists x \in R : xw \in L'\}$.*

Proposition 4.1.9 ([NO99])**.** *$\mathcal{L}$(det-R) is a quotient basis for the recursively enumerable languages.*

The next result is an immediate consequence of the latter proposition.

Proposition 4.1.10. *For each recursively enumerable language L, there is a* det-RWW*-transducer T such that L is the output language of T, that is, $ra(\mathrm{Rel}(T)) = L$.*

Proof. Here we use a slightly different perspective on Definition 4.1.8 also taken from [NO99]. A language class $\mathcal{C}$ is a quotient basis for the r.e. languages if, for each language $L \in \mathsf{RE}$ over some alphabet Σ, there exist a language $L' \in \mathcal{C}$ over some alphabet $\Sigma' \supset \Sigma$, a symbol $\# \in \Sigma'\backslash\Sigma$ and a regular set $R \subseteq (\Sigma'\backslash\{\#\})^*$ such that $L = \mathrm{cut}_{\#}(L' \cap R \cdot \# \cdot (\Sigma'\backslash\{\#\})^*)$. Here $\mathrm{cut}_{\#}$ is the operation which removes the prefix of a string x that ends with the unique occurrence of the symbol $\#$.

Observe that already a finite state transducer is able to compute the intersection with $R\cdot\#\cdot(\Sigma'\backslash\{\#\})^*$ and the $\mathsf{cut}_{\#}$-operation. Hence, it is easy to show that every r.e. language can be obtained by applying a finite state transduction on a language L' taken from a language class that is known to be a basis for RE.

Note that we here want to show a stronger result. Based on a language $L' \in$ det-R we show that every language $L \in \mathsf{RE}$ is the *output language* of a det-RWW-transducer. Next we describe this particular transducer.

Let M' be the det-R-automaton for L'. From M we define a det-RWW-automaton M'' that accepts the language $L'' = L' \cap R \cdot \# \cdot (\Sigma'\backslash\{\#\})^*$. Clearly this is possible as the

languages computed by det-RWW-automata are closed under taking intersections with regular languages (e.g. in [Ott06], p.10).

Due to the fact that a transducer for L'' must additionally compute the $\mathsf{cut}_{\#}$-operation, its transitions cannot be taken directly from M''. Actually we need to slightly adjust the language L''. Hence, from L'' we construct a language $L''_{\&}$ such that a word $w = w_1...w_n \in L''$ if and only if $w_{\&} = \&\&w_1\&\&w_2...\&\&w_n\&\& \in L''_{\&}$ $(w_1, w_2, ..., w_n \in \Sigma')$. Here & is a new symbol, not in Σ'. Obviously $L''_{\&} \in \mathcal{L}(\mathsf{det\text{-}RWW})$ holds[27]. Note that this special symbol is needed to proceed the $\mathsf{cut}_{\#}$-operation.

From the automaton M'' we now obtain the det-RWW-Td T, that is, every transition of M'' is a transition for T and all restart or accept steps of M'' are extended by the empty output. W.l.o.g assume that $k \geq 3$ is the window size of M''. Now the computation of T consists of three phases.

(1) First, T computes the $\mathsf{cut}_{\#}$-operation and it additionally verifies whether the input word w is correctly annotated by &&. For that, observe that T expects a word of the form

$$\&\&x_1\&\&x_2\&\& \dots x_k\&\&\#\&\&y_1\&\&y_2\&\& \dots y_l\&\&,$$

where $x_1, \dots, x_k, y_1, \dots y_l \in \Sigma'$ and $l, k \geq 0$ on the tape. Starting in its initial configuration, T rewrites every factor $\&\&x_i$ to $\&'x_i$, $\&\&\#$ to $\&'\#$, $\&\&\$$ to $\&'\$$ and $\&\&y_i$ to $\&'y_i$, where $\&'$ is a new auxiliary symbol. Additionally, it outputs every y_i during the latter rewrite steps. Observe that if the input w was of the form given above, then T has produced the output $y_1y_2 \dots y_l$. If not, it gets stuck.

(2) The second phase describes the behavior of T on

$$\&'x_1\&'x_2\&' \dots x_k\&'\#\&'y_1\&'y_2\&' \dots y_l\&',$$

which is the word obtained in the first phase. Note that a deterministic RWW-automaton is weakly monotone (e.g. in [Ott06], p.23). That is, informally speaking, a deterministic RWW-automaton cannot perform a rewrite step before it sees at least the first symbol of the previously rewritten substring in its read/write window. Thus, the auxiliary symbol $\&'$ is needed to "reset" the transducer onto the left end of the tape. However, after phase one is completed, T naturally shifts the window

[27] Roughly speaking a det-RWW-automaton for $L''_{\&}$ can be easily obtained from M'' by adding the transitions that delete an occurrence of && whenever it appears in the read/write-window.

to the right end of the tape. Hence, when seeing &′$ it observes that no sub-words && remain on the tape. So it proceeds as follows: beginning with &′$, T deletes every symbol &′ from right to left. For that, a sub-word &′$y_i y_{i+1}$ (respectively for all possible occurrence of x_i and #) is the unique indicator that &′ is the right most auxiliary symbol on the tape.

(3) Clearly phase two leads to a tape content of the form

$$x_1 x_2 \dots x_k \# y_1 y_2 \dots y_l.$$

Hence, we are back in the initial configuration of M'', the automaton for the language L''. From here on T acts like M''.

It is obvious how to derive the transition function from the description of T. Further, T accepts only words from the languages L'' or $L''_{\&}$. Observe that for every word $w \in L''$, T produces empty output and for every word $w_{\&} \in L''_{\&}$, T produces a word of the recursively enumerable language L. Hence, for every language $L \in \mathsf{RE}$, there is a det-RWW-Td such that L is the output language of this particular transducer. □

A similar result will not hold for det-R-transducers. For that observe that such a transducer is not capable of using auxiliary symbols. Hence, no information about the computation can be saved onto the tape. Therefore, it can be verified that the output language of such a transducer is in some sense "suffix-closed", that is, certain suffixes of a word w in the output language belong also to the output language. However, already the previous result implies that many closure properties and decision problems have to be answered negatively for transducers of the above type.

4.2 Monotone Restarting Transducers

According to the previous section, the general model of a restarting transducer is quite powerful. Therefore, we turn to more restricted versions. Here we study monotone restarting transducers. Observe that the notion of monotonicity naturally carries over from restarting automata. Hence, a restarting transducer is called monotone if every computation of the underlying restarting automaton is monotone (see Section 2.2, "Variants of Restarting Automata"). Further, recall from the Preliminaries that being monotone has a

major influence on the computational power of restarting automata. Thus, all monotone restarting automata that are also deterministic accept the deterministic context-free languages, and their non-deterministic counterparts form a hierarchy such that mon-RWW and mon-RRWW both accept the context-free languages. Hence, it is natural to ask whether monotonicity has the same influence on transducers, that is, are monotone restarting transducers somehow related to pushdown relations?

4.2.1 Upper Bound

To establish an upper bound for the computational power of monotone restarting transducers we return to the concepts of *proper*- and *input/output*-relations, introduced in Chapter 3. The following result forms the basis of our reflections (cf. Proposition 3.1.9 and Corollary 3.1.14):

$$\mathsf{PDR} = \begin{cases} \mathcal{R}\mathrm{el}_p(\mathsf{det\text{-}mon\text{-}R}) \\ \mathcal{R}\mathrm{el}_p(\mathsf{det\text{-}mon\text{-}RRWW}) \\ \mathcal{R}\mathrm{el}_{io}(\mathsf{mon\text{-}RWW}) \\ \mathcal{R}\mathrm{el}_{io}(\mathsf{mon\text{-}RRWW}). \end{cases}$$

In the following these facts will lead to a rough classification of the relations computed by monotone restarting transducers.

Proposition 4.2.1. *The class of relations that are computed by monotone restarting transducers of type X is included in the class of input/output-relations of monotone restarting automata of the same type X. Furthermore, this is also true for the corresponding deterministic versions of type X. Here $X = \{\mathsf{R},\ \mathsf{RR},\ \mathsf{RW},\ \mathsf{RRW},\ \mathsf{RWW},\ \mathsf{RRWW}\}$.*

Proof. [28]Let $T = (Q, \Sigma, \Delta, \Gamma, ¢, \$, q_0, k, \delta)$ be a mon-RRWW-Td that computes the relation $\mathrm{Rel}(T)$. Thus, a pair $(u, v) \in \Sigma^* \times \Delta^*$ belongs to $\mathrm{Rel}(T)$ if and only if there exists a computation of the form $(q_0 ¢ u \$, \varepsilon) \vdash_T^{c^*} (\mathsf{Accept}, v)$ (with $u \in \Sigma^*$ and $v \in \Delta^*$).

Without loss of generality we may assume that the output alphabet Δ is disjoint from the input alphabet Σ and the tape alphabet Γ. Further, T performs restart and accept instructions only on the \$-symbol.

[28]The proof is based on joined work with Friedrich Otto.

From here on the main idea is to define a mon-RRWW-automaton that accepts input words w that belong to the shuffle of u and v. Observe that u and v may not be simply concatenated, as the result derived from all pairs (u, v) may not necessarily be context free.

More formally we now construct a mon-RRWW-automaton $M = (Q, \Sigma', \Gamma', ¢, \$, q_0, k', \delta')$ by meta-instructions from a description of T by meta-instructions, where $\Sigma' = \Sigma \cup \Delta$, $\Gamma' = \Gamma \cup \Delta$ and the window size k' is the sum of the window size k and the longest output string z, which can be taken from the description of T. Further, $\mathsf{sh}(E_2, \Delta^*)$ denotes the shuffle of the languages $E_2 \subseteq \Gamma^*$ and Δ^*. Now each rewriting meta-instruction

$$(¢ \cdot E_1,\, x \to y,\, E_2 \cdot \$;\, z)$$

of T yields a rewriting meta-instructions

$$(¢ \cdot E_1,\, xz \to y,\, \mathsf{sh}(E_2, \Delta^*) \cdot \$)$$

for M, where $x, y \in \Gamma^*$, $z \in \Delta^*$. Finally, each accepting meta-instructions $(¢ \cdot E \cdot \$, \mathsf{Accept};\, z)$ of T leads to an accepting meta-instruction $(¢ \cdot E \cdot z \cdot \$, \mathsf{Accept})$ for M. Here observe that a mon-RRWW-Td is able to perform a non-monotone rewrite step in the tail of a computation, which is not exactly mirrored by accepting meta-instructions. Clearly this leads only to a minor adjustment in the construction above. If T performs such a non-monotone rewrite step in a tail, M will perform the same rewrite. Further, M scans the remainder of the tape, while expecting the output produced by T during the accept-step to the left of the \$-symbol.

Based on this description it is clear how to derive the transition function δ' of M from the transition function δ of T.

It remains to show that $\mathrm{Rel}(T) = \mathrm{Rel}_{io}(M)$ holds. Let $(u, v) \in \mathrm{Rel}(T)$, that is, there exists an accepting computation of T that consumes the input $u \in \Sigma^*$ and produces the output $v \in \Delta^*$. This computation consists of a sequence of cycles $C_1, C_2, ..., C_{m-1}$, where C_i $(1 \leq i \leq m-1)$ is of the form

$$\begin{aligned}&(q_0 ¢ u_i x_i w_i \$, v_i) \vdash^*_{\mathsf{MVR}} (¢ u_i q_i x_i w_i \$, v_i) \vdash^*_{\mathsf{Rewrite}} (¢ u_i y_i q_i' w_i \$, v_i)\\ &\vdash^*_{\mathsf{MVR}} (¢ u_i y_i w_i q_i'' \$, v_i) \vdash_{\mathsf{Restart}} (q_0 ¢ u_i y_i w_i \$, v_i z_i) = (q_0 ¢ u_{i+1} x_{i+1} w_{i+1} \$, v_{i+1})\end{aligned}$$

and a tail computation is of the form

$$(q_0\text{¢}\, u_m\$, v_m) \vdash^*_{\mathsf{MVR}} (\text{¢}\, u_m q_m\$, v_m) \vdash (\mathsf{Accept}, v_m z_m).$$

Obviously, for all $i = 1, ..., m-2$, $u_i y_i w_i = u_{i+1} x_{i+1} w_{i+1}$, and as T is monotone, we see that $|x_i w_i| \geq |x_{i+1} w_{i+1}|$ holds. This inequality further implies that w_{i+1} is a (not necessarily proper) suffix of w_i. Therefore, x_{i+1} can be decomposed to $\alpha_i y_i \beta_i$, where α_i is a suffix of u_i and β_i is a prefix of w_i.

Consequently, M simulates the computation of T as follows. It expects an input of the form $u_i x_i z_i w'_i$, where the current output z_i of T is inserted immediately to the right of the string x_i and $w'_i, w'_{i+1} \in \mathsf{sh}(w_i, \Delta^*)$. Now M will execute the following sequence of steps by using the meta-instructions that have been obtained from T:

$$\begin{aligned} &q_0\text{¢}\, u_i x_i z_i w'_i\$ \vdash^*_{\mathsf{MVR}} \text{¢}\, u_i q_i x_i z_i w'_i\$ \vdash^*_{\mathsf{Rewrite}} \text{¢}\, u_i y_i q'_i w'_i\$ \\ &\vdash^*_{\mathsf{MVR}} \text{¢}\, u_i y_i w'_i q''_i\$ \vdash_{\mathsf{Restart}} q_0\text{¢}\, u_i y_i w'_i\$ = q_0\text{¢}\, u_{i+1} x_{i+1} z_{i+1} w'_{i+1}\$ \end{aligned}.$$

Here, the interesting part is how the outputs z_i and z_{i+1} of the i-th and $i+1$-th cycle of T are composed in the restart configuration of the i-th cycle of M. For that notice again that $x_{i+1} = \alpha_i y_i \beta_i$, which is the part of the tape content that will be rewritten in the following cycle. Due to the fact that T is monotone, there are two different cases for the computation of T that M has to deal with.

Case 1: If $|x_i w_i| > |x_{i+1} w_{i+1}|$, that is, we are in a strictly monotone part of the computation, then w_{i+1} is a proper suffix of w_i and it follows that $|\beta_i| \geq 1$, and $|\alpha_i| \geq 0$. As β_i is a non-empty prefix of w_i, the restart configuration of the i'th cycle of M can be decomposed to

$$q_0\text{¢}\, u_{i+1} \alpha_i x_i z_i \beta_i z_{i+1} w'_{i+1}\$,$$

where $x_i z_i$ will be rewritten to y_i.

Case 2: The situation for $|x_i w_i| = |x_{i+1} w_{i+1}|$ is an immediate consequence of Case 1. Thus, $w_{i+1} = w_i$ and it follows that $|\beta_i| = 0$, and $|\alpha_i| \geq 1$. Then x_{i+1} is a suffix of $u_i y_i$ and the restart configuration of the i'th cycle of M can be decomposed to

$$q_0\text{¢}\, u_i x_i z_i z_{i+1} w'_{i+1}\$ = q_0\text{¢}\, u_{i+1} \alpha_i x_i z_i z_{i+1} w'_{i+1}\$,$$

where the current output z_i and the output of the next cycle of T z_{i+1} are inserted immediately to the right of the string x_i.

By combining these two cases we obtain a word $w \in \mathsf{sh}(u, v)$ such that the computation of M on input w mirrors the computation of T on input u and it follows that $(u, v) \in \mathrm{Rel}_{io}(M)$. Conversely, it is obvious that $(u, v) \in \mathrm{Rel}(T)$ holds for each pair $(u, v) \in \mathrm{Rel}_{io}(M)$. In addition, as this proof does not depend on auxiliary symbols, non-determinism or right computations, it holds for all types of monotone restarting automata. □

In the following the technique used to prove the last proposition will appear again to establish an upper bound for some simpler types of restarting transducers. Initially, the result puts all monotone machines in the context of pushdown relations.

Corollary 4.2.2. $\mathcal{R}\mathrm{el}(\mathsf{mon}\text{-}X\text{-}\mathsf{Td}) \subsetneq \mathsf{PDR}$ *and* $\mathcal{R}\mathrm{el}(\mathsf{det\text{-}mon}\text{-}X\text{-}\mathsf{Td}) \subsetneq \mathsf{PDR}$*, where* $X = \{\mathsf{R}, \mathsf{RR}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RWW}, \mathsf{RRWW}\}$.

The previous inclusions are actually proper for the reason that Proposition 4.1.4 ("length-bounded") does not hold for pushdown transducers in general, that is, a pushdown transducer is able to produce an arbitrarily long output on empty input.

4.2.2 Monotone Restarting Transducers and Pushdown Functions

In the following we focus on the question whether we can make the results derived by Proposition 4.2.1 more precise, that is, are there any other well-known classes within the pushdown relations that can be characterized in terms of restarting transducers? An overview of the most important subclasses of pushdown relations can be found in Section 2.3 (see "Pushdown Relations").

Initially, we focus on subclasses of pushdown relations that contain instances that can be accepted only by a pushdown transducer where the underlying pushdown automaton is non-deterministic. Obviously this is true for PDR, PDF and UPDF. Unfortunately, being functional or unambiguous is neither a syntactical property nor decidable for a pushdown transducer. Thus, a connection between these classes and common restarting transducers seems to be highly unlikely[29]. To verify this conjecture we consider the following example inspired by [JMPV99] (cf. Lemma 4.2).

[29] Certainly, and for further reflections we might investigate unambiguous or functional restarting transducers.

Example 4.2.3. Let

$$L_1 = \{f, fg\} \cdot \{a^n b^m c^m d^n \mid n, m \geq 0\},$$
$$L_2 = \{g, fg\} \cdot \{a^n b^n c^m d^m \mid n, m \geq 0\},$$

and $L = L_1 \cup L_2$ be a slightly different version of the context-free language $\{a^n b^m c^m d^n \mid n, m \geq 0\} \cup \{a^n b^n c^m d^m \mid n, m \geq 0\}$ that is well known for being inherently ambiguous (e.g. in [HU79]). Of course L is still context free and inherently ambiguous, as words of the sub-language $\{fg \cdot a^n b^n c^n d^n \mid n, m \geq 0\}$ lead to two different accepting computations. The semi-characteristic function χ'_L of L is defined as follows

$$\chi'_L(w) = \begin{cases} 1, & w \in L \\ \text{undefined}, & w \notin L \end{cases}.$$

Here we describe a mon-R-transducer $T = (Q, \{a, b, c, d, f, g\}, \{1\}, \Gamma, \mathfrak{c}, \$, q_0, 3, \delta)$ by meta-instructions that computes the function χ'_L:

$$\begin{array}{lllll} T: & (\mathfrak{c}, & fg \to f; & \varepsilon), & \\ & (\mathfrak{c}, & fg \to g; & \varepsilon), & \\ & (\mathfrak{c} \cdot f \cdot a^* b^*, & bcx \to x; & \varepsilon), & x \in \{c, d, \$\}, \\ & (\mathfrak{c} \cdot f \cdot a^*, & adx \to x; & \varepsilon), & x \in \{d, \$\}, \\ & (\mathfrak{c} \cdot f \cdot \$, & \mathsf{Accept}; & 1), & \\ & (\mathfrak{c} \cdot g \cdot a^*, & abx \to x; & \varepsilon), & x \in \{b, c, \$\}, \\ & (\mathfrak{c} \cdot g \cdot c^*, & cdx \to x; & \varepsilon), & x \in \{d, \$\}, \\ & (\mathfrak{c} \cdot g \cdot \$, & \mathsf{Accept}; & 1). & \end{array}$$

Thus, on input $fg \cdot a^* b^* c^* d^*$, T guesses non-deterministically whether the input word belongs to L_1 or L_2 and saves the guess by deleting one of the marker symbols f or g. From here on T deletes all factors bc or ab, respectively, then continues with ad or cd. T accepts and outputs the symbol 1 while seeing $\mathfrak{c} \cdot f \cdot \$$ or $\mathfrak{c} \cdot g \cdot \$$. Additionally, as T is not able to save information on the tape also the sub-languages $\{f \cdot a^n b^m c^m d^n \mid n, m \geq 0\}$ and $\{g \cdot a^n b^n c^m d^m \mid n, m \geq 0\}$ are accepted. Based on the description above it is clear that for every $w \in \Sigma^*$, $T(w) = \chi'_L(w)$ holds.

Example 4.2.3 shows that already monotone R-transducers can compute functions that are not unambiguous. Thus, we derive the following result.

Proposition 4.2.4. *The class of unambiguous pushdown functions (*UPDF*) and the relations computed by* mon-X*-transducers are incomparable under inclusion. Here X denotes transducers of type* $\{\mathsf{R}, \mathsf{RR}, \mathsf{RW}, \mathsf{RRW}\}$.

Proof. In Example 4.2.3 we presented a function that is computable by a mon-R-Td. As the set of input words of this particular function is known to be inherently ambiguous, there is no unambiguous pushdown transducer that computes this function.

Conversely in [JMPV99] it was shown that the context-free language $L = \{a^n b^n \mid n \geq 0\} \cup \{a^n b^m \mid m > 2n \geq 0\}$ is not accepted by any mon-RRW-automaton. Further, L is unambiguous. A pushdown automaton for L initially guesses whether the input is of the form $a^n b^n$ or $a^n b^m$ and then verifies its guess. Clearly such an automaton has a unique accepting computation for every word of the language L.

It follows that any relation, where the output depends on checking whether a word belongs to L is not computable by a mon-RRW-transducer, but there are functions of this form that are computable by an unambiguous pushdown transducer. □

Admittedly, from a computational point of view using Example 4.2.3 to prove the previous proposition was a bit arkward, as non-deterministic restarting transducers can compute relations that are not functional. Anyway, we know now that even the functions computed by restarting transducers of the latter type are incomparable to UPDF. The next result follows immediately from the hierarchy of pushdown relations.

Corollary 4.2.5. *The class of pushdown functions (*PDF*) and the relations computed by* mon-X*-transducers are incomparable under inclusion. Here X denotes transducers of type* $\{\mathsf{R}, \mathsf{RR}, \mathsf{RW}, \mathsf{RRW}\}$.

We can even strengthen the previous incomparability results to more general classes of restarting transducers. For that consider the following example.

Example 4.2.6. Let $\tau : \{a, b\}^* \mapsto \{a, b\}^*$ be a function that is defined by

$$\tau(u) = \begin{cases} \varepsilon & \text{, for } u = \varepsilon, \\ ba^n & \text{, for } u = ab^n \ (n \geq 0), \\ \text{undefined} & \text{, else.} \end{cases}$$

A sequential transducer for τ is shown in Figure 4.3.

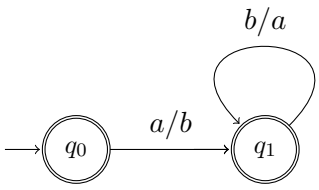

Figure 4.3: The sequential transducer from Example 4.2.6.

Proposition 4.2.7. $\mathsf{SeqF} \setminus \mathcal{R}\mathrm{el}(\mathsf{RRW\text{-}Td}) \neq \emptyset$

Proof. Let τ be the function from Example 4.2.6. Assume T is an RRW-transducer that computes τ. On input ab^n it must produce the output ba^n, that is, the first output produced in an accepting computation on input ab^n is of the form ba^i for some constant i. If n is large enough, then it is clear that T is not able to compute τ in a tail computation, that is, any accepting computation on a sufficiently long input word has at least one cycle. Due to the fact that an RRW-transducer cannot use auxiliary symbols and has to be correctness preserving (cf. Proposition 4.1.3), T must shorten the input word by deleting b's. Thus, after the first restart the current input word is ab^m, where $m < n$. From here on T must produce the output b^j, but that would violate the correctness preserving property for the reason that also pairs of words of the form (ab^n, bx) $(x \in \Delta^*)$ are computed by T. □

The latter results have shown that restarting transducer that are not capable of using auxiliary symbols are incomparable to most of the well-known relation classes. Therefore, we now turn to machines with an additional set of auxiliary symbols, that is, mon-RWW- and mon-RRWW-transducers. As stated above, there are pushdown relations that cannot be computed by any mon-RWW- (or mon-RRWW-)transducer and additionally, there are non-functional relations that can be computed by these devices. Thus, the question arises whether at least all pushdown functions can be computed by mon-RWW-Td or mon-RRWW-Td.

To answer the latter question we show an even stronger result. For that we define a new class of pushdown relations, called *length-bounded* PDR (lbPDR for short) in the sense of Definition 2.1.2.

Definition 4.2.8. *A pushdown relation R is length bounded if there is an integer $c \in \mathbb{N}$, such that for each pair $(u, v) \in R$ with $u \neq \varepsilon$, $|v| \leq c \cdot |u|$. The class of all length-bounded*

pushdown relations is denoted by lbPDR.

The following result is based on the notion of simple syntax directed translation schemes (sSDTS), which were introduced in Section 2.3 (see Definition 2.3.12) as the grammatical analogues to pushdown transducers.

Theorem 4.2.9. lbPDR $= \mathcal{R}$el(mon-RWW-Td)

Proof. Obviously the inclusion from right to left holds for the reason that a mon-RWW-Td can only compute PDR (cf. Proposition 4.2.1) and fulfills the length-bounded property (cf. Proposition 4.1.4). It remains to show that for any length-bounded pushdown relation $R \in$ lbPDR there is a mon-RWW-transducer T such that $R = \mathrm{Rel}(T)$.

In [JMPV99] Jancar et al. simulated a pushdown automaton by a mon-RWW-automaton. Here we extend their proof to transducers. To establish our construction we must use a pushdown transducer that is of the same special form as the pushdown automaton used in the original paper. That is:

(1) the pushdown transducer is non-deterministic and accepts by empty pushdown,

(2) before it increases the height of its pushdown by one symbol, it must read at least two input symbols,

(3) it reads at least one symbol in every step.

In terms of languages it is easy to verify that there is a pushdown automaton for any context-free language that fulfills the three conditions. This special machine is derived by using a grammar in Greibach normal form and standard compressing techniques to control the height of the pushdown store.

As it is not straightforward to verify that there is a pushdown transducer of the above type for any length-bounded pushdown relation, we first explain how a transducer of such a special form can be obtained.

By Proposition 2.3.13 it is clear that every pushdown relation can be defined by a syntax directed translation scheme S in quadratic Greibach normal form, that is, all rules are of the form $A \to (a\alpha, b\alpha)$, where a is a symbol of the input alphabet or the empty word, b is a symbol of the output alphabet or the empty word, a and b are not both the empty word and α is a string of non-terminals of length at most two.

Now let us assume that $S = (V, \Sigma, \Delta, P, S)$ is a sSDTS in quadratic Greibach normal form that generates a length-bounded relation. Thus, we can make some additional assumptions about the form of the rules in S. That is, any derivation that produces non-empty output on empty input has to be of bounded length. More formally, there is a positive integer k such that for any derivation of the form $(A, A) \Rightarrow_S^* (X_A, yX_A)$, where $A \in V$, $X_A \in V^*$ and $y \in \Delta^*$, the length of the output string y is bounded by k (and therefore also the number of non-terminals $|X_A|$), that is $|y| \leq k$. The latter statement holds for the reason that if there is no such integer k, this would violate the property of being length-bounded. From here on it is clear that for every $k \in \mathbb{N}$ there exists a sSDTS $S_1 = (V, \Sigma, \Delta, P_1, S)$ that is derived from S by eliminating rules of the form $A \to (\alpha, b\alpha)$. Thus P_1 contains all rules of P with the following exception:

- If $A \to (\alpha, b\alpha)$ is in P, then it is not in P_1, where $\alpha \in V \cup V^2$ and $b \in \Delta$.
- If there is a left most derivation of the form

$$(A, A) \Rightarrow_S^{k' \leq k} (X_A, yX_A) \Rightarrow_S (aBX'_A, ybBX'_A),$$

  then the rule $A \to (aBX'_A, ybBX'_A)$ is in P_1, where $A \in V$, $a \in \Sigma$, $b \in \Delta$, y is an output word, B is a non-terminal or empty and finally X'_A is a non-empty string of non-terminals.[30]

Admittedly, in a strict sense S_1 is not a syntax directed translation scheme, as here one derivation step possibly produces a string instead of one output symbol. Further, S_1 is not necessarily in quadratic Greibach normal form. Anyway, $\mathrm{Rel}(S) = \mathrm{Rel}(S_1)$ is easily proved if done by induction on the length of a derivation.

Continuing, let M_1 be a pushdown transducer for S_1, similar to the one exposed in e.g. [AU72], that simulates left most derivations of S_1 in its pushdown store. Thus, if M_1's topmost pushdown symbol is an A and the sSDTS S_1 has a rule of the form $A \to (a\alpha, y\alpha)$, then M_1 replaces A by α, checks whether the current input symbol is an a, and outputs y. Recall that $\alpha \in V^*$, a is a single input symbol and y is an output word or the empty word. Notice that M_1 already fulfills the conditions (1) and (3) from above and further on, it is clear that in any accepting computation the height of its pushdown store in step i is at most $k \cdot i$.

[30] Clearly the construction of S_1 depends on the knowledge of k.

From here on it is easy to see that M_1 can be transformed into a pushdown transducer M which uses a smaller pushdown store, that is, in step i the height of the pushdown store is at most $\frac{i}{2}+1$. This can be done by a standard compressing technique, where one pushdown symbol of M encodes $2k$ pushdown symbols of M_1. The machine M derived in this way also fulfills the condition (2) from above. Additionally notice that a pushdown transducer of this form will in general not exist for a pushdown relation that is not length bounded, as the height of the pushdown stack will not necessarily be bounded.

We have seen that for any length-bounded pushdown relation there is a pushdown transducer M of such a special form. Now M can easily be simulated by a mon-RWW-transducer T. For that the basic idea (taken from [JMPV99]) is that the current state of M and the content of the pushdown store is encoded on the tape of T to the left of the current input symbol of M. Without loss of generality the pushdown alphabet is disjoint from the input alphabet, i.e. T can distinguish a restarting configuration from an initial configuration. Finally one cycle of T corresponds to two steps of M, that is, two symbols are removed from the list and replaced by at most one pushdown symbol. During this step T produces the concatenated output that M has produced in its two consecutive steps.

Further, if a lbPDR also contains pairs of the form (ε, y), where $y \in \Delta^*$, then T can be modified to compute these pairs as follows; started on the empty tape T outputs y during the accept step. This completes the proof. □

Recall that we are looking for a classification of the relations computed by mon-RWW-Td among the pushdown functions. For that the latter result forms an appropriate basis, as it is clear that the class of single valued relations generated by sSDTS coincides with the class of pushdown functions. Here our first aim is to verify that there are no instances within the PDF that violate the length-bounded property. Therefore, we make use of the following properties of simple syntax directed translations, which were stated by Aho and Ullman.

Proposition 4.2.10 ([AU69]). *If R is a relation generated by an* sSDTS*, then there is a constant c, such that for all $u \neq \varepsilon$ in the domain of R, there is a v such that (u, v) is in R and $|v| \leq c \cdot |u|$.*

Let S be the syntax directed translation scheme (sSDTS) in Chomsky normal form that realizes R, thus $\mathrm{Rel}(S) = R$. The basic idea of the proof is to substitute all rules in S of the form $A \to (\varepsilon, b)$ (where A is a non-terminal and b is an output symbol) as they are

"responsible" for violating the length-bounded property. Finally a new sSDTS S' is derived that is not able to produce output strings of arbitrary length and by construction of S' it holds that $\mathrm{Rel}(S') \subseteq R$.

Here we omit further details on the proof and continue with an immediate consequence of the latter proposition.

Corollary 4.2.11 ([AU69]). *Let R be a single valued relation generated by an* sSDTS. *Then there is a constant c, such that if $u \neq \varepsilon$ and (u, v) is in R then $|v| \leq c \cdot |u|$.*

Thus, every single-valued relation generated by an sSDTS is length bounded. It follows that this is also true for the class of pushdown functions. Hence, from the previous interesting properties and Theorem 4.2.9 we can immediately obtain the following result.

Proposition 4.2.12. $\mathsf{PDF} \subsetneq \mathcal{R}\mathrm{el}(\mathsf{mon\text{-}RWW\text{-}Td})$

Proof. The inclusion from left to right is obvious for the reason that every pushdown function is length bounded (cf. Corollary 4.2.11). The properness of the inclusion can be derived from the fact that mon-RWW-Td can compute relations that are not functions (cf. Corollary 4.2.5). □

Moreover, as the relations computed by mon-RRWW-transducers are also included in the length-bounded pushdown relations the following consequence holds.

Corollary 4.2.13. $\mathcal{R}\mathrm{el}(\mathsf{mon\text{-}RRWW\text{-}Td}) = \mathcal{R}\mathrm{el}(\mathsf{mon\text{-}RWW\text{-}Td})$

We strongly expect that we can obtain a similar result for deterministic monotone restarting transducers in relation to the class DPDF. Further reflections on this suggestion can be found in the Open Questions Section.

4.3 Restarting Transducers with Window Size One

One of the main goals of the present work is to identify transductions that might be of practical interest. In this sense, relations computed by restarting transducers with window size one form an interesting topic on their own. It is well known (see Subsection 2.2.1, "Descriptional Complexity") that there are several types of restarting automata with window

size one, which characterize the regular languages, while offering quite succinct representations for some instances. This is a hint that there is not only a theoretical benefit in using restarting automata instead of finite state acceptors for practical purposes. Based on these observations it seems to be gainful to extend these results to transducers of the same type.

Therefore here we focus only on restarting transducers where the underlying automata characterize the regular languages. In Subsection 2.2.1 we presented some older results from Mráz and Reimann [Mrá01, Rei07]:

$$\mathcal{L}(\mathsf{det\text{-}R}(1)) = \mathcal{L}(\mathsf{mon\text{-}R}(1)) = \mathcal{L}(\mathsf{R}(1)) = \mathcal{L}(\mathsf{det\text{-}RR}(1)) = \mathsf{REG}.$$

There, also the following new characterizations of the regular languages by restarting automata can be found:

$$\mathcal{L}(\mathsf{det\text{-}mon\text{-}nf\text{-}R}(1)) = \mathcal{L}(\mathsf{mon\text{-}nf\text{-}R}(1)) = \mathcal{L}(\mathsf{det\text{-}mon\text{-}nf\text{-}RR}(1)) = \mathsf{REG}.$$

Especially the latter equivalences will lead to some interesting results for transducers.

4.3.1 Hierarchy Results

Unsurprisingly, the equivalences shown at the beginning of this chapter do not necessarily hold for transducers. In fact, we immediately obtain the following inclusions. Just observe that each det-R(1)- and each det-RR(1)-transducer is necessarily monotone (see Subsection 2.2.1, p.39).

Proposition 4.3.1.

(a)	$\mathcal{R}\mathrm{el}(\mathsf{det\text{-}R}(1)\text{-}\mathsf{Td})$	$\subseteq$	$\mathcal{R}\mathrm{el}(\mathsf{det\text{-}mon\text{-}nf\text{-}R}(1)\text{-}\mathsf{Td})$	$\subseteq$	$\mathcal{R}\mathrm{el}(\mathsf{det\text{-}mon\text{-}nf\text{-}RR}(1)\text{-}\mathsf{Td})$.
(b)	$\mathcal{R}\mathrm{el}(\mathsf{det\text{-}R}(1)\text{-}\mathsf{Td})$	$\subseteq$	$\mathcal{R}\mathrm{el}(\mathsf{det\text{-}RR}(1)\text{-}\mathsf{Td})$	$\subseteq$	$\mathcal{R}\mathrm{el}(\mathsf{det\text{-}mon\text{-}nf\text{-}RR}(1)\text{-}\mathsf{Td})$.
(c)	$\mathcal{R}\mathrm{el}(\mathsf{det\text{-}R}(1)\text{-}\mathsf{Td})$	$\subseteq$	$\mathcal{R}\mathrm{el}(\mathsf{det\text{-}mon\text{-}nf\text{-}R}(1)\text{-}\mathsf{Td})$	$\subseteq$	$\mathcal{R}\mathrm{el}(\mathsf{mon\text{-}nf\text{-}R}(1)\text{-}\mathsf{Td})$.
(d)	$\mathcal{R}\mathrm{el}(\mathsf{det\text{-}R}(1)\text{-}\mathsf{Td})$	$\subseteq$	$\mathcal{R}\mathrm{el}(\mathsf{mon\text{-}R}(1)\text{-}\mathsf{Td})$	$\subseteq$	$\mathcal{R}\mathrm{el}(\mathsf{R}(1)\text{-}\mathsf{Td})$.

Obviously, the above inclusions also hold for the corresponding types of *proper* transducers. Further, $\mathcal{R}\mathrm{el}(\mathsf{prop}\text{-}X) \subseteq \mathcal{R}\mathrm{el}(X)$ holds obviously for each type X of restarting transducers.

In fact, we can show that several of the inclusions above are actually strict. For that we use the following two examples.

Example 4.3.2. The function $\tau_1 : a^* \to \{b, c\}^*$ that is defined for $n \in \mathbb{N}$ by

$$\tau_1(a^n) = \begin{cases} b^n & ; \text{if } n \text{ is even,} \\ c^n & ; \text{if } n \text{ is odd,} \end{cases}$$

is computed by the *proper* det-mon-nf-RR(1)-transducer $T_1 = (Q, \{a\}, \{b, c\}, ¢, \$, q_0, 1, \delta)$ that is described by the following meta-instructions:

$$\begin{array}{llllll}
(1) & (q_0, & ¢, & a \to \varepsilon, & a \cdot (aa)^* \cdot \$, & q_1; \ b), \\
(2) & (q_0, & ¢, & a \to \varepsilon, & (aa)^* \cdot \$, & q_2; \ c), \\
(3) & (q_0, & ¢ \cdot \$, & \mathsf{Accept}; & & \varepsilon), \\
(4) & (q_1, & ¢, & a \to \varepsilon, & a^* \cdot \$, & q_1; \ b), \\
(5) & (q_1, & ¢ \cdot \$, & \mathsf{Accept}; & & \varepsilon), \\
(6) & (q_2, & ¢, & a \to \varepsilon, & a^* \cdot \$, & q_2; \ c), \\
(7) & (q_2, & ¢ \cdot \$, & \mathsf{Accept}; & & \varepsilon).
\end{array}$$

Example 4.3.3. The function $\tau_2 : \{0, 1\}^* \to \{0, 1\}^*$ that is defined by

$$\tau_2(w) = \begin{cases} 0^{|x|}, & \text{if } w = x0 \text{ and } x \in \{0, 1\}^*, \\ 1^{|x|}, & \text{if } w = x1 \text{ and } x \in \{0, 1\}^*, \end{cases}$$

is computed by the *proper* det-RR(1)-transducer $T_2 = (Q, \{0, 1\}, \{0, 1\}, ¢, \$, q_0, 1, \delta)$ that is described by the following meta-instructions, where $y \in \{0, 1\}$:

$$\begin{array}{llllll}
(1) & (¢, & y \to \varepsilon, & \{0,1\}^* \cdot 0 \cdot \$; & 0), \\
(2) & (¢, & y \to \varepsilon, & \{0,1\}^* \cdot 1 \cdot \$; & 1), \\
(3) & (¢ \cdot 0 \cdot \$, & \mathsf{Accept}; & & \varepsilon), \\
(3) & (¢ \cdot 1 \cdot \$, & \mathsf{Accept}; & & \varepsilon).
\end{array}$$

Proposition 4.3.4. *The function τ_1 can neither be computed by a* det-RR(1)*-, nor by an* R(1)*-, nor by a* det-mon-nf-R(1)*-transducer.*

Proof. Let T be an R(1)- or a det-RR(1)-transducer, and let n be a large positive integer. Then on input a^n, the transducer T cannot simply compute $\tau_1(a^n)$ in a tail computation. Hence, its accepting computation on input a^n begins with a cycle of the form $(q_0 ¢ a^n \$, \varepsilon) \vdash_T^c (q_0 ¢ a^{n-1} \$, v)$, where v is either of the form b^m or of the form c^m for a small value of m. If n is even, then $\tau_1(a^n) = b^n$, and if n is odd, then $\tau_1(a^n) = c^n$. Starting from the configuration

$(q_0\text{¢}\, a^{n-1}\$, v)$, T computes the word $v \cdot \tau_1(a^{n-1})$. However, from the definition of τ_1 it follows immediately that $\tau_1(a^n) \neq v \cdot \tau_1(a^{n-1})$. Thus, τ_1 is not computed by any R(1)- or det-RR(1)-transducer.

Finally, let T be a det-mon-nf-R(1)-transducer. The accepting computation of T on input a^n consists of a finite sequence of cycles that is followed by an accepting tail computation. As T restarts immediately on executing a delete operation, it can read the end marker \$ only in a tail computation. Thus, during the above sequence of cycles T does not see the input a^n completely, and so it cannot produce any non-empty output. However, during the accepting tail computation it cannot produce the complete output, if n is large. Thus, it follows that τ_1 is not computed by any det-mon-nf-R(1)-transducer, either. □

Proposition 4.3.5. *The function τ_2 can neither be computed by an* R(1)- *nor by a* det-mon-nf-R(1)-*transducer.*

Proof. As in the proof of Proposition 4.3.4 it follows that a det-mon-nf-R(1)-transducer cannot compute the function τ_2, as it does not see its input completely before it executes a tail computation. On the other hand, if T is an R(1)-transducer for computing τ_2, then on input 0^n0 it must produce the output 0^n, while on input 0^n1 it must produce the output 1^n. As in an accepting tail computation T can only produce an output of fixed finite length, each accepting computation of T on input 0^n0 has the form

$$(q_0\text{¢}\, 0^n0\$, \varepsilon) \vdash_T^{c^i} (q_0\text{¢}\, 0^{n+1-i}\$, 0^j) \vdash_T^* (\mathsf{Accept}, 0^n),$$

where the i-th cycle is the first one in which a non-empty output is produced. Then, $0 < i < c$ and $0 < j < c$ hold for some constant c. However, T would then also execute the accepting computation

$$(q_0\text{¢}\, 0^n0 \cdot 1\$, \varepsilon) \vdash_T^{c^i} (q_0\text{¢}\, 0^{n+1-i}1\$, 0^j) \vdash_T^* (\mathsf{Accept}, 0^j1^{n+1-i}),$$

which shows that T does not compute the function τ_2. □

From these propositions and Proposition 4.2.7 we obtain the hierarchy shown in Figure 4.4, where τ_3 is the function described in Proposition 4.2.7. Here the properness of the inclusions of the deterministic classes and the corresponding non-deterministic classes follows from the simple fact that deterministic transducers can only compute functions, while non-

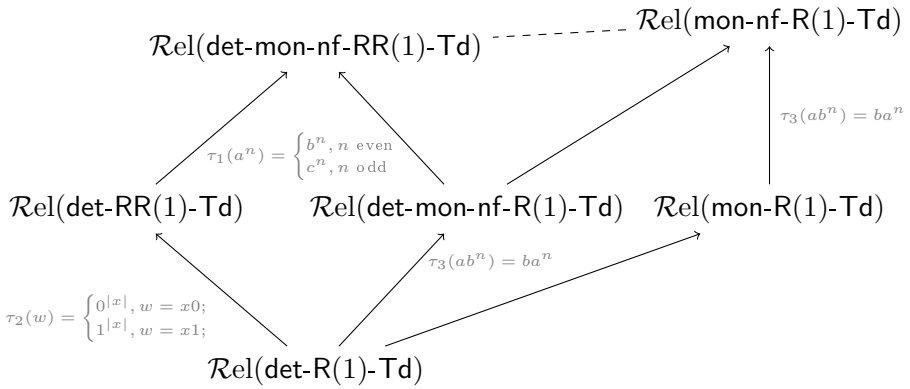

Figure 4.4: Hierarchy of classes of transductions computed by various types of restarting transducers with window size one. Here arrows denote proper inclusions, dashed lines denote unknown relationships, and no arrows denote incomparability.

deterministic transducers can also compute relations that are not functions. Additionally, incomparability results can easily be derived as a combination of the latter results.

Finally, the class $\mathcal{R}$el(R(1)-Td) is not taken into account yet, as up to now we are not able to derive some proper results on it. Later we will see that this particular class is for some reason of its own interest.

4.3.2 Characterizing Classes of Rational Relations

To relate our transducers to well-known classes of transductions, we again use the technique based on the input/output relations, which has been applied to prove that the monotone restarting transducers are included in the pushdown relations (cf. Proposition 4.2.1). In the following this will lead to an upper bound for all monotone types of transducers with window size one.

In Chapter 3 it was shown that the class of input/output-relations ($\mathcal{R}\text{el}_{io}$) of any type of restarting automaton that characterizes the regular languages, coincides with the rational relations (RAT). Thus, if we are able to adapt Proposition 4.2.1 to restarting transducers with window size one, we put the relations of these machines in the context of rational relations.

Proposition 4.3.6. $\mathcal{R}\mathrm{el}(\mathsf{det\text{-}mon\text{-}nf\text{-}RR(1)\text{-}Td}) \subsetneq \mathcal{R}\mathrm{el}_{io}(\mathsf{det\text{-}mon\text{-}nf\text{-}RR(1)})$

Proof. The proposition is proved in quite the same way as the corresponding result for monotone transducers with an arbitrary window size. There, the automaton, which simulates the transducer, accepts a language where the output of the transducer is shuffled into the input, such that in every rewrite step of the automaton a part of the input and the corresponding output occurs in the window. Thus, one cycle of the automaton mirrors one cycle of the transducer. Hence, if the window size is restricted to one, then the situation is slightly more complicated, as the latter idea does not work anymore. However, we can fix this problem by using the non-forgetting property.

Accordingly, let $T = (Q, \Sigma, \Delta, \mathfrak{c}, \$, q_0, 1, \delta)$ be a $\mathsf{det\text{-}mon\text{-}nf\text{-}RR(1)}$-transducer that computes a relation $\mathrm{Rel}(T) \subseteq \Sigma^* \times \Delta^*$. Thus, a pair $(u, v) \in \Sigma^* \times \Delta^*$ belongs to $\mathrm{Rel}(T)$ if and only if there exists a computation of the form $(q_0 \mathfrak{c} u \$, \varepsilon) \vdash_T^{c^*} (q_i \mathfrak{c} u' \$, v') \vdash_T^* (\mathsf{Accept}, v'v'')$ such that $v = v'v''$.

Without loss of generality we can assume that T performs restart and accept instructions only on the \$-symbol. Also we may assume that Σ and Δ are disjoint. We now define a $\mathsf{det\text{-}mon\text{-}nf\text{-}RR(1)}$-automaton M by meta-instructions from a description of T by meta-instructions. In contrast to the situation described in Proposition 4.2.1 here each rewriting meta-instruction of T is translated into finitely many rewriting meta-instructions of M. However, in order to increase readability we just consider the case that the output b produced in a single step is of length at most one, but this construction is easily extended to the case of output words of any positive length by using additional rewriting meta-instructions and additional restart states.

Let $(q_i, E_1, a \to \varepsilon, E_2 \cdot \$, q_j; b)$ be a rewriting meta-instruction of T, where $a \in \Sigma$ and $b \in \Delta$. It is translated into the rewriting meta-instructions

$$(q_i, E_1, a \to \varepsilon, \mathsf{sh}(E_2, \Delta^*) \cdot \$, q_i') \text{ and } (q_i', E_1, b \to \varepsilon, \mathsf{sh}(E_2, \Delta^*) \cdot \$, q_j)$$

of M, where q_i' is a new state. If $b = \varepsilon$, that is, the rewriting meta-instruction of T considered is of the form $(q_i, E_1, a \to \varepsilon, E_2 \cdot \$, q_j; \varepsilon)$, then we simply take the rewriting meta-instruction $(q_i, E_1, a \to \varepsilon, \mathsf{sh}(E_2, \Delta^*) \cdot \$, q_j)$ for M. Finally, each accepting meta-instruction $(q_i, E \cdot \$, \mathsf{Accept}; b)$ of T yields an accepting meta-instruction $(q_i, E \cdot b\$, \mathsf{Accept})$ of M. Based on this description the transition function of M can be derived from the transition function of T.

It remains to show that $\mathrm{Rel}(T) = \mathrm{Rel}_{\mathrm{io}}(M)$ holds. Let $(u, v) \in \mathrm{Rel}(T)$, that is, there exists an accepting computation of T that consumes input $u \in \Sigma^*$ and produces output $v \in \Delta^*$. This computation consists of a sequence of cycles $C_1, C_2, \ldots, C_{m-1}$, where C_i $(1 \leq i \leq m-1)$ is of the form

$$\begin{aligned}&(q_i\mathfrak{c}x_ia_iy_i\$, v_i) \vdash^*_{\mathsf{MVR}} (\mathfrak{c}x_ip_ia_iy_i\$, v_i) \vdash_{\mathsf{Delete}} (\mathfrak{c}x_ip'_iy_i\$, v_i)\\ &\vdash^*_{\mathsf{MVR}} (\mathfrak{c}x_iy_i\hat{p}_i\$, v_i) \vdash_{\mathsf{Restart}} (q_{i+1}\mathfrak{c}x_iy_i\$, v_ib_i),\end{aligned}$$

and a tail computation of the form

$$(q_m\mathfrak{c}w_m\$, v_m) \vdash^*_{\mathsf{MVR}} (\mathfrak{c}w_mq'_m\$, v_m) \vdash_{\mathsf{Accept}} (\mathsf{Accept}, v_mb').$$

In the above cycle a meta-instruction $(q_i, E_1, a_i \to \varepsilon, E_2\$, q_{i+1}; b_i)$ is applied, where $\mathfrak{c}x_i \in E_1$ and $y_i \in E_2$, and in the above tail computation an accepting meta-instruction $(q_m, E \cdot \$, \mathsf{Accept}; b')$ is applied, where $\mathfrak{c}w_m \in E$ holds. Obviously, for all $i = 1, \ldots, m-2$, $x_iy_i = x_{i+1}a_{i+1}y_{i+1}$, and as T is monotone, we see that $|y_i| \geq |y_{i+1}|$ holds.

M simulates the above cycle of T as already presented in Proposition 4.2.1. However, for the sake of completeness we will again outline the main idea.

Case 1. If $|y_i| > |y_{i+1}|$, then $a_{i+1}y_{i+1}$ is a suffix of y_i. Thus, if we insert the letter b_i immediately to the right of the letter a_i, then M will execute the following sequence of two cycles using the meta-instructions that have been obtained from the above meta-instruction of T, where y'_i is from the shuffle of y_i with a word from Δ^*:

$$\begin{aligned}(q_i\mathfrak{c}x_ia_ib_iy'_i\$) &\vdash^*_{\mathsf{MVR}} (\mathfrak{c}x_ip_ia_ib_iy'_i\$) &&\vdash_{\mathsf{Delete}} (\mathfrak{c}x_ip'_ib_iy'_i\$)\\ &\vdash^*_{\mathsf{MVR}} (\mathfrak{c}x_ib_iy'_i\hat{p}_i\$) &&\vdash_{\mathsf{Restart}} (q'_i\mathfrak{c}x_ib_iy'_i\$)\\ &\vdash^*_{\mathsf{MVR}} (\mathfrak{c}x_ip''_ib_iy'_i\$) &&\vdash_{\mathsf{Delete}} (\mathfrak{c}x_i\tilde{p}_iy'_i\$)\\ &\vdash^*_{\mathsf{MVR}} (\mathfrak{c}x_iy'_i\hat{p}'_i\$) &&\vdash_{\mathsf{Restart}} (q_{i+1}\mathfrak{c}x_iy'_i\$).\end{aligned}$$

Case 2. If $|y_i| = |y_{i+1}|$, then $y_i = y_{i+1}$ and $x_i = x_{i+1}a_{i+1}$. In this situation we insert the word b_ib_{i+1} immediately to the right of the factor $a_{i+1}a_i$. Then M will execute the following sequence of two cycles using the meta-instructions that have been obtained from the above meta-instruction of T, where y'_i is from the shuffle of y_i with a word from Δ^*:

$$
\begin{array}{lll}
(q_i\text{¢}\, x_i a_i b_i b_{i+1} y_i'\$) & \vdash^*_{\mathsf{MVR}} (\text{¢}\, x_i p_i a_i b_i b_{i+1} y_i'\$) & \vdash_{\mathsf{Delete}} (\text{¢}\, x_i p_i' b_i b_{i+1} y_i'\$) \\
& \vdash^*_{\mathsf{MVR}} (\text{¢}\, x_i b_i b_{i+1} y_i' \hat{p}_i\$) & \vdash_{\mathsf{Restart}} (q_i'\text{¢}\, x_i b_i b_{i+1} y_i'\$) \\
& \vdash^*_{\mathsf{MVR}} (\text{¢}\, x_i p_i'' b_i b_{i+1} y_i'\$) & \vdash_{\mathsf{Delete}} (\text{¢}\, x_i \tilde{p}_i b_{i+1} y_i'\$) \\
& \vdash^*_{\mathsf{MVR}} (\text{¢}\, x_i b_{i+1} y_i' \hat{p}_i'\$) & \vdash_{\mathsf{Restart}} (q_{i+1}\text{¢}\, x_i b_{i+1} y_i'\$) \\
& & = (q_{i+1}\text{¢}\, x_{i+1} a_{i+1} b_{i+1} y_{i+1}'\$).
\end{array}
$$

By combining these two cases we obtain a word $w \in \mathsf{sh}(u, v)$ such that the computation of M on input w mirrors the computation of T on input u, and it follows that $(u, v) \in \mathrm{Rel}_{\mathrm{io}}(M)$. Conversely, it can be checked easily that $(x, y) \in \mathrm{Rel}(T)$ holds for each pair $(x, y) \in \mathrm{Rel}_{\mathrm{io}}(M)$. Thus, $\mathrm{Rel}(T) = \mathrm{Rel}_{\mathrm{io}}(M)$ follows. In addition, as T is deterministic and monotone, so is M.

Finally the inclusion is proper for the fact that Corollary 4.1.5 holds for restarting transducers and the relation presented there is obviously rational. □

Analogously also the following inclusion can be derived.

Lemma 4.3.7. $\mathcal{R}\mathrm{el}(\mathsf{mon\text{-}nf\text{-}R(1)\text{-}Td}) \subsetneq \mathcal{R}\mathrm{el}_{io}(\mathsf{mon\text{-}nf\text{-}R(1)})$.

Up to now we have seen that most of the inclusions given in Proposition 4.3.1 collapse into the rational relations. In summary, together with Proposition 3.1.8 from Chapter 3 the latter facts yield the following results.

Corollary 4.3.8.

$$
\begin{array}{lll}
\text{(a)} & \mathcal{R}\mathrm{el}(\mathsf{det\text{-}mon\text{-}nf\text{-}RR(1)\text{-}Td}) & \subseteq \ \mathsf{RATF}. \\
\text{(b)} & \mathcal{R}\mathrm{el}(\mathsf{mon\text{-}nf\text{-}R(1)\text{-}Td}) & \subsetneq \ \mathsf{RAT}.
\end{array}
$$

Observe that (b) is a strict inclusion for the reason that RAT contains relations that are not length bounded in the sense of Proposition 4.1.4.

Thus, we have obtained numerous relation classes within the rational relations that are characterized by restarting transducers. Naturally, the question arises whether (a) is a strict inclusion and secondly, whether there are any restarting transducer characterizations for some traditional subclasses of the rational relations and/or rational functions, which were introduced in the Preliminaries.

Our first characterization result shows that the dGSM-mappings correspond to a particular class of restarting transducers.

Theorem 4.3.9. *A function $f : \Sigma^* \to \Delta^*$ is a* dGSM*-function if and only if it can be computed by a proper* det-mon-nf-R(1)*-transducer.*

Proof. Let $D = (Q, \Sigma, \Delta, \delta, q_0, F)$ be a dGSM. We define a det-mon-nf-R(1)-transducer $M = (Q, \Sigma, \Delta, ¢, \$, q_0, 1, \delta)$ such that $\mathrm{Rel}(M) = \mathrm{Rel}(D)$ holds. The transducer M is obtained from D by converting every transition step $(q, x) \to (p, y)$ ($q, p \in Q$, $x \in \Sigma$, and $y \in \Delta^*$) of D into the transition steps $\delta(q, ¢) = (q, \mathsf{MVR})$ and $\delta(q, x) = (p, \mathsf{Restart}, y)$[31]. As M is an R(1)-transducer, its restart operations are combined with delete operations. Thus, M simulates D by erasing its tape inscription letter by letter from left to right, for each letter producing the corresponding output. Finally, M accepts restarting from the restarting configuration $(q¢\$, w)$ producing the empty output if and only if q is a final state of D. It follows that $\mathrm{Rel}(M) = \mathrm{Rel}(D)$, and that M is proper, monotone, and deterministic.

Conversely let M be a proper det-mon-nf-R(1)-transducer that computes a transduction $t : \Sigma^* \to \Delta^*$. In Theorem 2.2.11 it is shown that each det-mon-nf-R(1)-automaton can be simulated by a deterministic finite-state acceptor (DFA). During the simulation the DFA has to store a bounded number of possible delete/restart operations of the restarting automaton in its finite-state control in order to verify that it has detected a correct sequence of cycles within the computation being simulated. Now, by storing the possible output word together with each delete/restart operation, a dGSM can be designed that simulates the transducer M.

More formally, let $D = (Q_D, \Sigma, \Delta, \delta_D, q_0^{(D)}, F)$ be the dGSM that simulates the restarting transducer M. The description of D is taken from the description of the corresponding DFA given in the proof of Theorem 2.2.11. Thus, the set of states is defined as

$$Q_D = \{[q_r, T, B] \mid q_r \in Q, \text{ and } q_r \text{ is a restart state of } M\},$$

where T is the state table that mirrors the computations that correspond to the different restart states in parallel, and B is the buffer defined as a matrix in which possible sequences of restarts are saved. Recall that the entries of B are tuples that mirror restarting steps. Here we extend these tuples by an additional component that contains the corresponding output word, that is, we now have tuples of the form $(a, (q, p, q'), v)$. This tuple records a possible cycle of M, in which M, starting from the restart state q, performs a number

[31] Note, to increase readability we use this slightly different representation for restarting transitions, which were introduced in Subsection 2.2.1 in the form of $\delta(q, x) = (p, \varepsilon, y)$.

of move-right steps until it reaches state p, and when reading the symbol a in state p, M executes a delete/restart step which takes it to the restart state q' while producing output v. Based on the proof of Theorem 2.2.11, it is quite clear that the dGSM D is able to produce the correct output for a given input. For example, let

$$(a_i, (q_{j_{i-1}}, p_i, q_{j_i}), v_i), \ldots, (a_1, (q_{j_0}, p_1, q_{j_1}), v_1), (a_0, (q_j, p_0, q_{j_0}), v_0)$$

be a sequence of restart steps of M that is saved in the buffer B, and let $\delta(p_{i+1}, a_{i+1}) = (q_j, \mathsf{Restart}, v')$ be the current step that D wants to simulate, which completes a cycle of M that started in state q_r. As the next cycle of M begins in state q_j, D realizes that M actually performed a sequence of $i+2$ cycles during which it produced the output $v'v_0v_1\cdots v_i$. Accordingly, in the next step D removes this sequence from the buffer B and produces the complete output $v'v_0v_1\cdots v_i$. Of course, in all previous steps of D during which the above sequence was stored in B, the dGSM just produced the empty output. Continuing in this way, it follows that $\mathrm{Rel}(D) = \mathrm{Rel}(M)$. □

If the given det-mon-nf-R(1)-transducer T is not proper, that is, if it produces non-empty outputs during some of its accept transitions, then the above construction yields a subsequential transducer. On the other hand, it can easily be seen that a subsequential transducer can be simulated by a det-mon-nf-R(1)-transducer that is allowed to produce non-empty outputs during its accept instructions. Thus, we have the following consequence.

Corollary 4.3.10. *A function $f : \Sigma^* \to \Delta^*$ is a subsequential function if and only if it can be computed by a* det-mon-nf-R(1)*-transducer.*

Further, as the GSM is the non-deterministic version of the dGSM also the following result holds.

Theorem 4.3.11. *A relation $R \subseteq \Sigma^* \times \Delta^*$ is a* GSM*-relation if and only if it can be computed by a proper* mon-nf-R(1)*-transducer.*

Proof. Based on the proof of Theorem 2.2.12, the proof of Theorem 4.3.9 easily extends to mon-nf-R(1)-transducers. □

Finally, we want to characterize the class of rational functions in terms of restarting transducers. To this end we need the following result of Santean [San04].

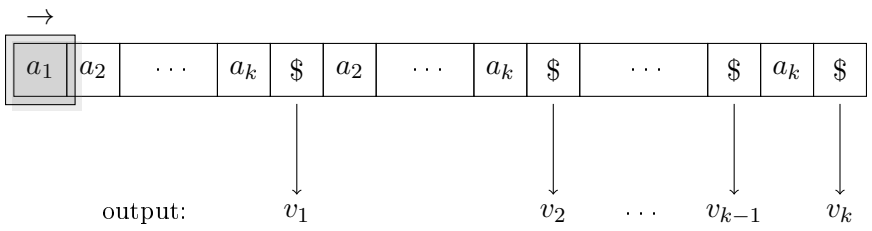

Figure 4.5: Sketch of the sequential transducer that simulates a bimachine.

Here $\mu_\$: \Sigma^* \to (\Sigma \cup \{\$\})^*$ denotes the function defined by

$$\mu_\$(\varepsilon) = \varepsilon \text{ and } \mu_\$(a_1 \dots a_k) = a_1 \dots a_k \$ a_2 \dots a_k \$ \dots \$ a_{k-1} a_k \$ a_k \$,$$

for $k \geq 1$ and $a_1, \dots, a_k \in \Sigma$.

Theorem 4.3.12 ([San04]). *If $f : \Sigma^* \to \Delta^*$ is a rational function such that $f(\varepsilon) = \varepsilon$, then there exists a sequential function $f_L : (\Sigma \cup \{\$\})^* \to \Delta^*$ such that $f = f_L \circ \mu_\$$.*

Proof. Here we only want to explain the main idea of Santeans proof. Basically it is based on a result on bimachines presented in the same paper. Recall that a bimachine is a finite automaton with two heads. One scans the tape from left to right and the other scans the tape in the opposite direction. The output is produced when both heads meet at the current input symbol (see Section 2.3). However, it can be shown that the scanning direction of the heads of a bimachine is not of importance as long as the machine has seen the whole input before outputting the corresponding symbols.

Hence, from a bimachine a sequential transducer can be obtained that computes the same function while working on an extended input word given by $\mu_\$$. Figure 4.5 illustrates the sequential transducer used to compute the given rational function. There it can be seen that the transducer scans the input completely and produces its output only while seeing the \$-symbol. □

Of course, $\mu_\$$ is not rational, and in fact, it is not even a pushdown function. However, the restarting transducers are somehow naturally equipped to simulate this preprocessing stage.

Theorem 4.3.13. RATF $\subseteq \mathcal{R}$el(det-mon-nf-RR(1)-Td).

Proof. Let $f : \Sigma^* \to \Delta^*$ be a rational function. Let us first assume that $f(\varepsilon) = \varepsilon$ holds. By Theorem 4.3.12 there exists a sequential function $f_L : (\Sigma \cup \{\$\})^* \to \Delta^*$ such that $f = f_L \circ \mu_\$$. As the function f_L is sequential, it can be computed by a proper det-mon-nf-RR(1)-transducer T (cf. Theorem 4.3.9).

Now this transducer can be extended to a det-mon-nf-RR(1)-transducer T_f for computing f. The sequential transducer for f_L that is given in the proof of Theorem 4.3.12 produces a non-empty output only on seeing the \$-symbol. Now T_f proceeds as follows. During the first cycle on input $u = a_1 \dots a_k$, it erases the letter a_1 and simulates the internal transitions of the sequential transducer for f_L until it reaches the \$-symbol. At this time it restarts and produces the corresponding output. Now the next cycle starts with the tape content $a_2 \dots a_k$. Continuing in this way $f(u) = f_L \circ \mu_\$(u)$ is computed. Thus, T_f is a proper det-mon-nf-RR(1)-transducer that computes the function f.

Finally, if $f(\varepsilon) \neq \varepsilon$, then we apply the construction above to the partial function f' that is defined by $f'(u) = f(u)$ for all $u \in \Sigma^+$ and $f'(\varepsilon) = \varepsilon$. This yields a proper det-mon-nf-RR(1)-transducer T'_f for computing f'. We then extend T'_f such that, starting from its initial state, it accepts on empty input producing the output $f(\varepsilon)$. □

Together with Corollary 4.3.8 (a) this yields the main result in this section.

Corollary 4.3.14. RATF $= \mathcal{R}$el(det-mon-nf-RR(1)-Td).

We conclude this subsection by answering the open question whether $\mathcal{R}$el(det-mon-nf-RR(1)-Td) is included in $\mathcal{R}$el(mon-nf-R(1)-Td) (see Figure 4.4). From Theorem 4.3.11 we know that the classes GSMRel and $\mathcal{R}$el(prop-mon-nf-R(1)-Td) coincide. Further, in Section 2.3 (see p.63) we discussed the fact that every rational function τ, with $\tau(\varepsilon) = \varepsilon$, is computable by a GSM and thus, by a proper mon-nf-R(1)-transducer. Then, clearly, there is a non-proper mon-nf-R(1)-transducer for every rational function, particularly for those that contain a mapping from the empty word ε to an arbitrary word v over the output alphabet. This, and the fact that mon-nf-R(1)-transducer compute non-functional relations, immediately yield the following proper inclusion.

Corollary 4.3.15. $\mathcal{R}$el(det-mon-nf-RR(1)-Td) $\subsetneq \mathcal{R}$el(mon-nf-R(1)-Td).

4.3.3 Summary

In the previous two subsections we compared the relations computed by restarting transducers, where the underlying automata characterize the regular languages, with each other and also to traditional relation classes. In this context $\mathcal{R}\mathrm{el}(\mathsf{R}(1)\text{-}\mathsf{Td})$ forms an exception. Somehow surprisingly, this particular class contains relations that are not rational. For that consider the following example.

Example 4.3.16. Let $T = (Q, \{a, b, \#\}, \{a, b\}, ¢, \$, q_0, 1, \delta)$ be the $\mathsf{R}(1)$-transducer that is described by the following meta-instructions:

$$\begin{array}{lllll} (1) & (¢, & \# \to \varepsilon; & \varepsilon), \\ (2) & (¢, & b \to \varepsilon; & b), \\ (3) & (¢ \cdot \$, & \mathsf{Accept}; & \varepsilon), \\ (4) & (¢ \cdot \# \cdot b^*, & a \to \varepsilon; & a). \end{array}$$

T computes the relation $\mathrm{Rel}(T) = R \subseteq \{a, b, \#\}^* \times \{a, b\}^*$.

It might be possible to give an exact description for R. However, for our purposes the following result suffices.

Lemma 4.3.17. $T(\{\# \cdot (ab)^n \mid n \geq 0\}) = \{a^n b^n \mid n \geq 0\}$.

Proof. Let $u = \# \cdot (ab)^n$ be an input word for the restarting transducer T of Example 4.3.16, where $n \geq 0$. It remains to show that $T(\# \cdot (ab)^n) = a^n b^n$.

Assume that $n \geq 1$, as $n = 0$ implies that in an accepting computation only the meta-instructions (1) and (3) from the example above are applicable. Thus T produces the wanted output $v = \varepsilon$. When starting in the configuration $(q_0 ¢ \# \cdot (ab)^n \$, \varepsilon)$ $(n \geq 1)$, there are two meta-instructions of T that can be used to begin the computation. Here we want to list all possibilities and the corresponding consequences. Applying meta-instruction:

(1) This leads to the configuration $(q_0 ¢ (ab)^n \$, \varepsilon)$. From here on there is no way to delete any symbol a, thus T can never reach an accepting configuration.

(4) Meta-instruction (4) can be applied as long as there are a's on the tape. This leads to the configuration $(q_0 ¢ \# \cdot b^k (ab)^{n-k} \$, a^k)$, where $1 \leq k \leq n$. Further on, for any $k < n$ there exists the possibility to use meta-instruction (1), but then we are in the situation described before. Hence, no accepting configuration is reachable.

For $k = n$, applying (1) leads to the configuration $(q_0 ¢ b^n \$, a^n)$. The computation can only continue with using (2). However, this leads to the configuration $(q_0 ¢ \$, a^n b^n)$, and so by (3), T accepts the input and produces the output $v = a^n b^n$.

These are all possible computations on input $u = \# \cdot (ab)^n$, and thus it is shown that $T(\{\# \cdot (ab)^n \mid n \geq 0\}) = \{a^n b^n \mid n \geq 0\}$. □

Proposition 4.3.18. $\mathcal{R}$el(R(1)-Td) *is incomparable to* RAT *with respect to inclusion.*

Proof. In Lemma 4.3.17 it was shown that there exists a R(1)-transducer that computes the transduction $T(\{\# \cdot (ab)^n \mid n \geq 0\}) = \{a^n b^n \mid n \geq 0\}$. Obviously $\{\# \cdot (ab)^n \mid n \geq 0\}$ is a regular language, while $\{a^n b^n \mid n \geq 0\}$ is context free. From Proposition 2.3.16 we know that rational transductions preserve regular languages. Hence T is not a rational transduction, that is, there exist relations that are computable by an R(1)-Td but not by a finite state transducer.

Conversely there are rational relations, more precisely sequential functions, that are not computable by R(1)-Td. This was shown in Proposition 4.2.7. □

We strongly suspect that the latter incomparability result carries over even to pushdown relations. For that consider an R(1)-Td T' that realizes the transduction $T'(\{\# \cdot (abc)^n \mid n \geq 0\}) = \{a^n b^n c^n \mid n \geq 0\}$. Clearly if T' exists, then by Proposition 2.3.18 we derive the incomparability to pushdown transductions. Further, it might be interesting to compare R(1)-transducers to *two-way finite state transducers* (e.g. exposed in [EY71]), as both machines show a similar behavior. A discussion on this topic can be found in the Open Question Section.

In conclusion Figure 4.6 gives an overview on the results of Section 4.3.

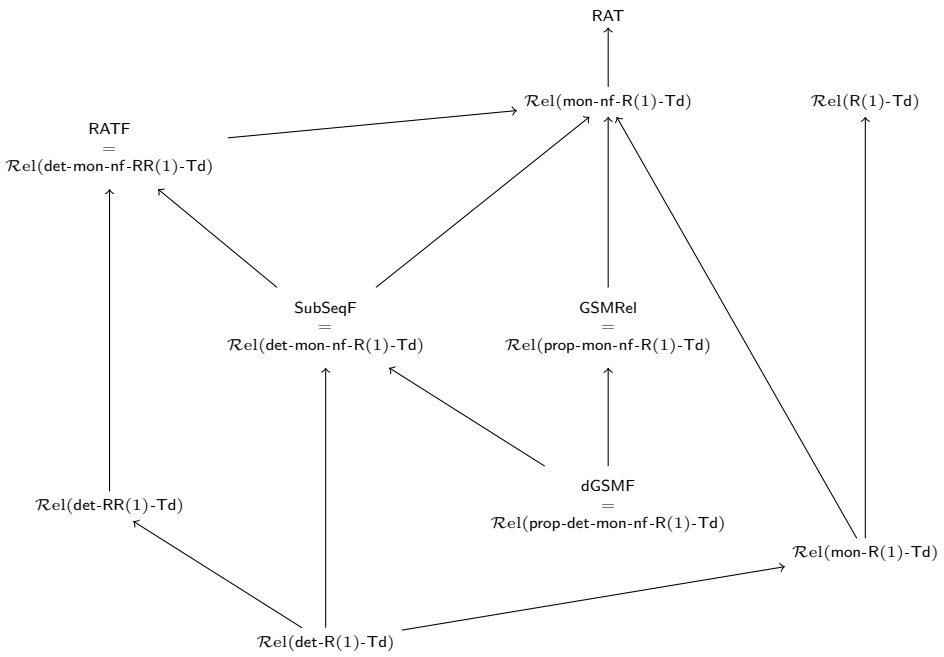

Figure 4.6: Summarized hierarchy of the classes of relations computed by the various restarting transducers with window size one. Here arrows denote proper inclusions, and no arrows denote incomparability.

4.4 Closure Properties

This section is not meant to be a comprehensive study on the closure properties of restarting transducers. We rather want to point out the most interesting property.

Closure under Composition

We stated already in the Preliminaries Section that the composition operation is of major importance for transductions in practical applications. Here we mainly focus on restarting transducers with window size one. The reason for that is the following proposition.

Proposition 4.4.1. *The relations computed by proper* det-mon-R-*transducers are not closed under composition.*

Proof. Here we mainly follow the well-known fact that the pushdown relations are not closed under composition. Let

$$T_1 = (Q_1, \{a, b, c, \hat{a}, \#\}, \{a, b, c, \hat{b}, \#\}, \Sigma, ¢, \$, q_0, 4, \delta_1) \text{ and}$$
$$T_2 = (Q_2, \{a, b, c, \hat{b}, \#\}, \{a, b, c\}, \Sigma, ¢, \$, p_0, 3, \delta_2)$$

be two det-mon-R-Td that are described by meta-instructions as follows:

$$\begin{array}{lllll}
T_1: & (1) & (¢, & \#\hat{a}ax \to \hat{a}ax; & \#), \quad x \in \{\hat{a}, b\}, \\
& (2) & (¢ \cdot a^*, & \hat{a}a\hat{a} \to a\hat{a}; & a), \\
& (3) & (¢ \cdot a^*, & \hat{a}ab \to ab; & a), \\
& (4) & (¢ \cdot a^*, & abx \to x; & \hat{b}b), \quad x \in \{b, c\}, \\
& (5) & (¢, & cx \to x; & c), \quad x \in \{c, \$\}, \\
& (6) & (¢ \cdot \$, & \mathsf{Accept}; & \varepsilon).
\end{array}$$

$$\begin{array}{lllll}
T_2: & (1) & (¢ \cdot \#, & a \to \varepsilon; & a), \\
& (2) & (¢ \cdot \# \cdot b^*, & \hat{b}b\hat{b} \to b\hat{b}; & b), \\
& (3) & (¢ \cdot \# \cdot b^*, & \hat{b}bc \to bc; & b), \\
& (4) & (¢ \cdot \# \cdot b^*, & bcx \to x; & c), \quad x \in \{c, \$\}, \\
& (5) & (¢ \cdot \# \cdot \$, & \mathsf{Accept}; & \varepsilon).
\end{array}$$

Next we describe the behavior of T_1. If an input word starts with the prefix $\#\hat{a}a$ then, by the meta-instructions (1), (2) and (3) T_1 enables a computation, where every $\hat{a}$ is followed

by an a and additionally T_1 produces the symbol # and an a for every $\hat{a}$. The computation of T_1 continues with meta-instruction (4), where the transducer checks if the remaining prefix of the input word is in the form of $a^n b^n$. Further it outputs the word $\hat{b}b$ for every subword ab on the tape. Finally, every c on the tape leads to a c in the output language (meta-instructions (5) and (6)). Hence, it is clear that, among others, $\mathrm{Rel}(T_1) = R_1$ contains the following subset:

$$R_1' = \{(\#(\hat{a}a)^n b^n c^m, \#a^n(\hat{b}b)^n c^m) \mid n, m \geq 1\}.$$

Observe that there is no other subset in R_1 that contains a pair in the form of $(u, \#v)$.

The behavior of T_2 is similar. First of all T_2 only accepts words that starts with the #-symbol. Then, it deletes every occuring a and produces an a (1). The remaining tape content of T_2 must be in the form of $\# \cdot b^* \cdot (\hat{b}b)^* \cdot c^*$. Thus, T_2 continues like T_1, that is, one b is produced for every $\hat{b}$ ((2) and (3)) and finally, one c is produced for every bc on the tape ((4) and (5)). Again, these meta-instructions ensure that a subset of the relation $\mathrm{Rel}(T_2) = R_2$ is:

$$R_2' = \{(\#a^m(\hat{b}b)^n c^n, a^m b^n c^n) \mid n, m \geq 0\}.$$

Note that the somehow artificial symbols $\hat{a}$ and $\hat{b}$ are needed for the reason that transducers of type R are not able to compute an "identity" mapping and verify the correctness of the input at the same time. This is also the reason for the fact that the languages accepted by the underlying automata of T_1 and T_2 have this somehow fuzzy structure. However, it is clear from the descriptions of T_1 and T_2 that the composition of both transducers leads to the following relation:

$$R = R_2 \circ R_1 = \{(\#(\hat{a}a)^n b^n c^n, a^n b^n c^n) \mid n \geq 1\}.$$

In particular, T_2 only enables computations on input words that starts with #. The #-symbol is the prefix of an output word of T_1 if and only if T_1 computes a pair from R_1'. Hence, it is clear that R only contains the shown pairs. R is clearly not a pushdown relation, as its range is a context-sensitive language. Further, by Corollary 4.2.2 it is also not a relation computed by any proper det-mon-R-transducer. □

Moreover, the latter result immediately extends to all kinds of restarting transducers, for which it is known that their class of relations is a superclass of $\mathcal{R}$el(prop-det-mon-R-Td) and which is included in the class of pushdown relations.

Corollary 4.4.2. *The classes of relations defined by (proper) (deterministic) monotone restarting transducers are not closed under composition.*

As already mentioned in Subsection 4.1.1, more general types of restarting transducers (i.e. non-monotone) seem to be way too powerful to establish nice closure properties. Therefore, we omit further discussions on these types of machines, here. Instead, we turn directly to restarting transducers with window size one. Especially we focus on det-R(1)-Td, mon-R(1)-Td and det-RR(1)-Td for the reason that all other types somehow coincide with some well-known transducer models. Obviously, all these types of restarting transducers inherit the closure properties of their classical counterparts.

Unfortunately, we will see next that mon-R(1)-Td, det-RR(1)-Td and det-R(1)-Td do not provide the property of being closed under composition.

Proposition 4.4.3. *The class of relations computed by* mon-R(1)*-transducers is not closed under composition.*

Proof. Here we give an example of two mon-R(1)-transducers, such that their composition is not a mon-R(1)-Td. For that $T_1 = (Q_1, \{0,1\}, \{0,1\}, ¢, \$, q_0^{(1)}, 1, \delta_1)$ and $T_2 = (Q_2, \{0,1\}, \{0,1\}, ¢, \$, q_0^{(2)}, 1, \delta_2)$ are defined by meta-instructions as follows:

$$
\begin{array}{lllll}
T_1: & (¢, & \{0,1\} \to \varepsilon; & 0), \\
 & (¢, & \{0,1\} \to \varepsilon; & 1), \\
 & (¢ \cdot 0 \cdot \$, & \mathsf{Accept}; & 0), \\
 & (¢ \cdot 1 \cdot \$, & \mathsf{Accept}; & 1), \\
 & & & \\
T_2: & (¢ \cdot 0, & \{0,1\} \to \varepsilon, & 0), \\
 & (¢ \cdot 1, & \{0,1\} \to \varepsilon, & 1), \\
 & (¢ \cdot 11 \cdot \$, & \mathsf{Accept}; & 1), \\
 & (¢ \cdot 00 \cdot \$, & \mathsf{Accept}; & 0).
\end{array}
$$

It is easy to verify that T_1 just maps the input word onto an arbitrary output word of the same length such that the last letter of the input equals the last letter of the output, that is,

$$R_1 = \mathrm{Rel}(T_1) = \{(ux, vx) \mid u, v \in \{0,1\}^*,\ x \in \{0,1\} \text{ and } |u| = |v|\}.$$

T_2's behavior is a little bit more involved. It outputs a word over 0 (respective 1) that is one letter shorter as the input word if and only if the first and the last letter of the input

coincide. Thus,

$$R_2 = \mathrm{Rel}(T_2) = \{(xwx, x^{|w|+1}) \mid w \in \{0,1\}^*,\, x \in \{0,1\}\}.$$

Next we claim that the composition of both is

$$R = R_2 \circ R_1 = \{(wx, x^{|w|}) \mid w \in \{0,1\}^*\}.$$

Let us show the following equivalence exemplary for the letter 0: a pair $(z \cdot 0, z' \cdot 0) \in R_1$ and $(z' \cdot 0, 0^{|z'|}) \in R_2$ if and only if $(z \cdot 0, 0^{|z|}) \in R$. For that assume that the input word z is of length n. Actually z can be mapped under T_1 onto exponential many outputs (with respect to n) z' and so there are these numbers of pairs of the form $(z \cdot 0, z' \cdot 0)$ in R_1. Hence, all possible $z' \cdot 0$ are the input of T_2. Now T_2 only accepts inputs where the first and last letter coincide and so all the possible outputs of $z \cdot 0$ under T_2 are split into two sets, the set of accepted and the set of non-accepted inputs of T_2. Finally, T_2 obviously accepts all words of the form $z' \cdot 0$, where $z' = 0 \cdot u$ and maps them all onto the same word $0^{|z'|}$. As the length of z' equals the length of z, we have the pair $(z \cdot 0, 0^{|z|})$ pair in R_2. Obviously the converse direction can be shown in the same way.

However, by Proposition 4.3.5 we know that R is not even in $\mathcal{R}\mathrm{el}(\mathsf{R}(1)\text{-}\mathsf{Td})$, so this completes the proof. □

Proposition 4.4.4. *The class of relations computed by* det-RR(1)*-transducers is not closed under composition.*

Proof. Consider the following det-RR(1)-transducers $T_1 = (Q_1, \{a\}, \{b,c\}, ¢, \$, q_0^{(1)}, 1, \delta_1)$ and $T_2 = (Q_2, \{b,c\}, \{b,c\}, ¢, \$, q_0^{(2)}, 1, \delta_2)$. Both are defined by the following meta-instructions:

$$\begin{array}{lllll} T_1: & (¢, & a \to \varepsilon, & a \cdot (aa)^*; & b), \\ & (¢, & a \to \varepsilon, & (aa)^*; & c), \\ & (¢ \cdot \$, & \mathsf{Accept}; & & \varepsilon). \end{array}$$

$$\begin{array}{lllll} T_2: & (¢ \cdot b, & \{b,c\} \to \varepsilon, & \{b,c\}^*; & b), \\ & (¢ \cdot c, & \{b,c\} \to \varepsilon, & \{b,c\}^*; & c), \\ & (¢ \cdot b \cdot \$, & \mathsf{Accept}; & & b), \\ & (¢ \cdot c \cdot \$, & \mathsf{Accept}; & & c). \end{array}$$

Clearly T_1 maps an even number of a's to an output word that begins with b and an odd number of a's to an output word that begins with c. In particular

$$R_1 = \mathrm{Rel}(T_1) = \{(a^{2n}, (bc)^n), (a^{2n+1}, (cb)^n c) \mid n \geq 0\}.$$

In contrast T_2 outputs a word over b (or c) according to the first letter of the input. Equally to T_1 the output word is of the same length as the input.

Hence,

$$R_2 = \mathrm{Rel}(T_2) = \{(xw, x^{|w|+1}) \mid x \in \{b, c\},\, w \in \{b, c\}^*\}.$$

Next consider the composition of both machines, that is, $R_2 \circ R_1$. As mentioned T_1 outputs a string beginning with b (respective c) if the input word over a had an even (odd, respectively) length . Hence, when T_2 computes this output, it maps the output of T_1 onto a word over b (respective c), according to the first letter. Thus, the composed relation is

$$R_2 \circ R_1 = R = \{(a^{2n}, b^{2n}),\, (a^{2n+1}, c^{2n+1}) \mid n \geq 0\},$$

which is not a det-RR(1)-transducer's relation (cf. Proposition 4.3.4). □

Proposition 4.4.5. *The class of relations computed by* det-R(1)*-transducers is not closed under composition.*

Proof. Here we give an example of two det-R(1)-transducers, such that their composition is not a det-R(1)-Td. For that $T_1 = (Q_1, \{a, b\}, \{c_1, c_2\}, ¢, \$, q_0^{(1)}, 1, \delta_1)$ and $T_2 = (Q_2, \{c_1, c_2\}, \{a, b\}, ¢, \$, q_0^{(2)}, 1, \delta_2)$ are defined by meta-instructions as follows:

$$\begin{array}{lllll}
T_1: & (¢ \cdot a, & b \to \varepsilon; & c_1 c_2), \\
 & (¢ \cdot a \cdot \$, & \mathsf{Accept}; & \varepsilon), \\
 & & & \\
T_2: & (¢ \cdot c_1, & c_2 \to \varepsilon, & ba), \\
 & (¢ \cdot c_1 c_1, & c_2 \to \varepsilon, & a), \\
 & (¢ \cdot c_1 c_1, & c_1 \to \varepsilon, & \varepsilon), \\
 & (¢ \cdot c_1 c_1 \cdot \$, & \mathsf{Accept}; & \varepsilon).
\end{array}$$

Clearly T_1 computes the relation

$$R_1 = \mathrm{Rel}(T_1) = \{(ab^n, (c_1 c_2)^n) \mid n \geq 0\}.$$

Further, on input $(c_1c_2)^*$, T_2 is only able to apply the first meta-instruction, that is, it deletes the first c_2 and produces ba. From that on, the prefix of the input is of the form c_1c_1 and thus, only the last three instructions are applicable. Hence, T_2 maps every further c_2 to a and c_1 to ε. Then, obviously

$$T_2(\{(c_1c_2)^n\}) = \{ba^n\}$$

and with $R_2 = \mathrm{Rel}(T_2)$ we have

$$R = R_2 \circ R_1 = \{(ab^n, ba^n) \mid n \geq 0\},$$

which is known to be not computable by any RRW-transducer (cf. Proposition 4.2.7.). □

Observe that we actually used a proper det-R(1)-transducer in the latter proof and that Proposition 4.4.3 and 4.4.4 can be adjusted in the same way. Thus, being non-proper is not the reason why these simple types of machines are not closed under composition.

However, we will show next what causes the previous results, at least for det-R(1)-transducers. For that we start our investigation by introducing a new model for computing transductions.

Definition 4.4.6. *A deterministic generalized sequential machine $M = (Q, \Sigma, \Delta, \delta, q_0, F)$ is called output-forgetting (*of-dGSM *for short) if for every transition of the form $\delta(q, a) = (p, \beta)$, where $p, q \in Q$, $a \in \Sigma$ and $\beta \in \Delta^+$, $q = p$ holds. Hence, an* of-dGSM *is a* dGSM *that is not allowed to change the state while producing a non-empty output.*

Lemma 4.4.7. $\mathcal{R}\mathrm{el}(\text{of-dGSM}) = \mathcal{R}\mathrm{el}(\text{prop-det-R(1)-Td})$

Proof. Let $M = (Q, \Sigma, \Delta, \delta, q_0, F)$ be a of-dGSM. From the description of M we build a proper det-R(1)-Td $T = (Q, \Sigma, \Delta, ¢, \$, q_0, 1, \delta')$ such that $\mathrm{Rel}(M) = \mathrm{Rel}(T)$. First recall that every computation of a det-R(1)-transducer is strictly monotone (see Subsection 2.2.1). This means that in each sequence of configurations the right distance is always shortened due to the fact that this machine is deterministic. Secondly, an of-dGSM M is fully defined by three kinds of transitions, that are, either M changes the state, then $\delta(q, a) = (p, \varepsilon)$, or it does not change the internal state, then $\delta(q, a) = (q, \beta)$ or $\delta(q, a) = (q, \varepsilon)$.

Now for every $p, q \in Q$, $a \in \Sigma$ and $\beta \in \Delta^+$, the transition function of T is described as follows:

- If $\delta(q,a) = (p,\varepsilon)$, then $\delta'(q,a) = (p, \mathsf{MVR})$.
- If $\delta(q,a) = (q,\beta)$, then $\delta'(q,a) = (\mathsf{Restart}, \beta)$.
- If $\delta(q,a) = (q,\varepsilon)$, then $\delta'(q,a) = (\mathsf{Restart}, \varepsilon)$.
- For every $q \in F$ we add a transition $\delta'(q,\$) = (\mathsf{Accept}, \varepsilon)$.
- As the restarting transducer begins every computation with its head over the ¢-symbol we finally add $\delta'(q_0, ¢) = (q_0, \mathsf{MVR})$.

Hence, T is fully defined and it remains to show that $\mathrm{Rel}(M) = \mathrm{Rel}(T)$.

For an input word $u \in \Sigma^*$, let (q_0u, ε) be the initial configuration of M. Accordingly, T starts with the configuration $(q_0¢u\$, \varepsilon)$, and it performs a move-right step that leads to $(¢q_0u\$, \varepsilon)$. T is now in the configuration that corresponds to the initial configuration of M.

From here on the computation of M consists of a sequence of computation steps where for every step, there are three cases according to the transition function δ. Next these three cases are described for the i'th step of the sequential machine. Hence, M is in the configuration $(u_1u_2...u_{i-1}q_iu_i...u_n, v_1v_2...v_m)$ $(n > 0,\ m \geq 0)$, where all indexed letters u are from the input alphabet and all indexed letters v are words over the output alphabet. Furthermore, assume that T is in the corresponding configuration, that is, $(¢u'q_iu_i...u_n\$, v_1v_2...v_m)$, where u' is possibly a scattered sub-word of $u_1u_2 \cdots u_{i-1}$.

Case 1: There is a transition of the form $\delta(q_i, u_i) = (q_{i+1}, \varepsilon)$, where $q_i \neq q_{i+1}$. Then, the computation step of M is

$$(u_1u_2...q_iu_i...u_n, v_1v_2...v_m) \vdash_M (u_1u_2...u_iq_{i+1}u_{i+1}...u_n, v_1v_2...v_m).$$

Hence the restarting transducer performs the following step:

$$(¢u'q_iu_i...u_n\$, v_1v_2...v_m) \vdash_T^{\mathsf{MVR}} (¢u'u_iq_{i+1}u_{i+1}...u_n\$, v_1v_2...v_m).$$

Case 2: There is a transition of the form $\delta(q_i, u_i) = (q_i, v_{m+1})$. Then, the computation step of M is

$$(u_1u_2...q_iu_i...u_n, v_1v_2...v_m) \vdash_M (u_1u_2...u_iq_iu_{i+1}...u_n, v_1v_2...v_mv_{m+1}).$$

To mirror this step, T first performs a restart-step, that is,

$$(\¢\, u' q_i u_i ... u_n \$, v_1 v_2 ... v_m) \vdash_T^{\mathsf{Restart}} (q_0 \¢\, u' u_{i+1} ... u_n \$, v_1 v_2 ... v_m v_{m+1}).$$

Now to make sure that this restarting configuration leads to the corresponding configuration of M, we have to explain how T behaves on the prefix u'. As mentioned u' is a scattered sub-word of $u_1 u_2 \cdots u_{l-1} u_l u_{l+1} \cdots u_k u_{k+1} \cdots u_{i-1}$, that is, possible substrings $u_l \cdots u_k$ are cut out by restart-steps of T. Now according to T's transition function, letters can only be deleted if and only if the of-dGSM performs a step without changing the state. Thus, on the substring $u_{l-1} u_l \cdots u_k u_{k+1}$, M applied the transition steps $\delta(q_{l-1}, u_{l-1}) = (q_l, \varepsilon)$, $\delta(q_l, u_j) = (q_l, v_i)$ $(l \leq j \leq k)$ and $\delta(q_l, u_{k+1}) = (q_{k+2}, \varepsilon)$, where all indexed q's are states from Q, and v_i are output words that occur consecutively $k - l$ times in $v_1 \cdots v_m$. Clearly δ' has in this case the following MVR-instructions: $\delta'(q_{l-1}, u_{l-1}) = (q_l, \mathsf{MVR})$ and $\delta'(q_l, u_{k+1}) = (q_{k+2}, \mathsf{MVR})$. Hence, started in the restarting configuration above, T performs the following computational steps:

$$(q_0 \¢\, u' u_{i+1} ... u_n \$, v_1 v_2 ... v_m v_{m+1}) \vdash_T^{\mathsf{MVR}^*} (\¢\, u' q_{i+1} u_{i+1} ... u_n \$, v_1 v_2 ... v_m v_{m+1}).$$

This means that it reaches a configuration that corresponds to the current configuration of the of-dGSM. Further on, if M accepts, that is, while reading the last letter from the tape, M switches into an accepting state, then T performs an additional move-right step and accepts while seeing the \$-symbol. In conclusion it is shown that there is a det-R(1)-transducer T for every of-dGSM M such that $\mathrm{Rel}(M) = \mathrm{Rel}(T)$.

Conversely, let $T = (Q, \Sigma, \Delta, \¢, \$, q_0, 1, \delta)$ be a proper det-R(1)-transducer. W.l.o.g assume that T only accepts while seeing the \$-symbol. Hence, it is clear how a dGSM $M = (Q, \Sigma, \Delta, \delta', q_0, F)$, has to be build, such that $\mathrm{Rel}(T) = \mathrm{Rel}(M)$ holds. For that recall again that a det-R(1)-transducer is necessarily monotone (see Subsection 2.2.1). Then the transition function δ' of M can be taken quite direct from the transition function δ of T. This, for instance, can be found in [Mrá01] or [Rei07]. Furthermore, when T restarts, M will not change its internal state. Thus, we derive a dGSM that is obviously *output-forgetting*.

Again the overall construction of M is fundamentally based on the following observation. Whenever a rewrite step occurs, then, as T is deterministic, in the following cycle it reaches exactly the letter next to the deleted letter in exactly the same state. According to the construction, this corresponds to a non-empty output transition where no change of state

happens. The proof of correctness is obvious and so clearly there is a of-dGSM M for every det-R(1)-transducer T such that $\mathrm{Rel}(T) = \mathrm{Rel}(M)$. □

We already know from Proposition 4.4.5 that proper det-R(1)-transducers are not closed under composition. It follows that of-dGSMs are not closed under composition, either. The reason for that is the possibillity to output more than one letter in each step, which will be shown by the next result.

Theorem 4.4.8. *The class of relations defined by* of-dGSM *that are only able to produce single-letter outputs is closed under composition.*

Proof. We here apply the standard cross-product technique exposed for instance in [Moh97]. Let $M_1 = (Q_1, \Sigma, \Gamma, \delta_1, q_0^{(1)}, F_1)$ and $M_2 = (Q_2, \Gamma, \Delta, \delta_2, q_0^{(2)}, F_2)$ be two of-dGSM with $R_1 = \mathrm{Rel}(M_1)$ and $R_2 = \mathrm{Rel}(M_2)$. We want to show that $R_2 \circ R_1$ is again an of-dGSM-relation. For that let $M = (Q_1 \times Q_2, \Sigma, \Delta, \delta, \langle q_0^{(1)}, q_0^{(2)} \rangle, F_1 \times F_2)$ be a new machine where for every $q^{(1)} \in Q_1$, $q^{(2)} \in Q_2$ and $a \in \Sigma$, δ is defined as follows:

$$\delta\left(\langle q^{(1)}, q^{(2)} \rangle, a\right) = \left(\left\langle \delta_1^s(q^{(1)}, a), \delta_2^s(q^{(2)}, \delta_1^o(q^{(1)}, a)) \right\rangle, \delta_2^o(q^{(2)}, \delta_1^o(q^{(1)}, a))\right),$$

Here δ_1^s, δ_1^o (for δ_2 respectively) are used to increase readability, that is, the upper s denotes the mapping onto the state and the upper o onto the output component of the transition function. Further, if $\delta_1^o(q^{(1)}, a) = \varepsilon$, that is, M_1 produces empty output, then $\delta_2^s(q^{(2)}, \delta_1^o(q^{(1)}, a)) = q^{(2)}$. The reason for that is, M_2 is not able to perform ε-steps and therefore whenever M_1 produces empty output M_2 will not change its configuration.

Clearly and according to the literature the so defined transducer realizes the composition of M_1 and M_2, that is, $\mathrm{Rel}(M) = R_2 \circ R_1$. It remains to show that M still is an of-dGSM, that is, it does not change the state while producing non-empty output. For that observe that from a dynamical point of view M_2 only produces non-empty output if M_1 has done, too. Thus, let $\delta_1(q, a) = (q, b)$, that is, while scanning an a in state q, M_1 produces a b. Then this b is the input of the second machine M_2. Hence, there are two possible steps for M_2 on b: $\delta_2(p, b) = (p, c)$ or $\delta_2(p, b) = (p', \varepsilon)$, where p, p' are states and c is an arbitrary output. Obviously only the first transition leads also to an output of M, that is, M performs the following combined step $\delta(\langle q, p \rangle, a) = (\langle q, p \rangle, c)$, where no change of state happens. Thus, it is clear that M still is an of-dGSM. □

An immediate consequence of the equivalence of proper det-R(1)-transducers and of-dGSM and Theorem 4.4.8 is the next corollary.

Corollary 4.4.9. *The class of relations defined by* prop-det-R(1)*-transducers that are only able to produce single-letter outputs is closed under composition.*

Clearly we might extend the latter results to non-proper det-R(1)-transducers by considering *subsequential transducers* instead of dGSMs. We omit further extensions for the reason that the functions computed by det-R(1)-transducers with single-letter output are fairly simple, regardless the condition of being proper or non-proper. However, we assume that these machines are able to compute every dGSMF up to a morphism, that is, the morphism is used to add auxiliary symbols to every word in the domain of a dGSMF. Whenever a dGSM changes its state while producing non-empty output, an of-dGSM expects an input that is annotated by auxiliary symbols such that this machine is able to change its internal state to the state of the dGSM, when scanning the auxiliary symbols.

4.5 Decision Problems

This last section in Chapter 4 should be seen more as a starting point for further reflections than as a complete investigation of decision problems. Thus, we shortly address one trivial and one more involved result on the two propably most important decision problems for transducers.

The Equivalence Problem:

Instance: Given two restarting transducers T_1 and T_2.

Question: Is $\mathrm{Rel}(T_1) = \mathrm{Rel}(T_2)$?

Secondly we introduce a not so well-known problem unique for transducers, which concerns practical purposes (eg. exposed in [FRR$^+$10, RS08]).

The Type-Checking Problem:

Instance: Given a restarting transducer T and two languages L_1 from the language class $\mathcal{C}_1$ and L_2 from the language class $\mathcal{C}_2$.

Question: Is $T(L_1) \subseteq L_2$?

The Equivalence Problem for single-valued finite state transducers is decidable (see Section 2.3) and Corollary 4.3.14 together with Theorem 4.3.9 yield an effective construction of a single-valued finite state transducer from a det-mon-nf-RR(1)-transducer. Therefore the following result holds.

Corollary 4.5.1. *The Equivalence Problem for* det-$X(1)$*-transducer is decidable, where* $X \in \{\mathsf{R}, \mathsf{RR}, \mathsf{mon\text{-}nf\text{-}R}, \mathsf{mon\text{-}nf\text{-}RR}\}$.

Now we turn to the so called Type-Checking Problem. Clearly an interesting property, especially for transducers that are used in practical applications. For instance, deterministic pushdown transducers are common in the field of compiler construction and therefore, decidability of type-checking against context-free languages seems a worthy question. Although it is not clear whether deterministic and monotone restarting transducers form a subclass of deterministic pushdown transducers, they are for sure related. Hence, type-checking might also be of interest for these machines.

Proposition 4.5.2. *The Type-Checking Problem for* det-mon-X*-transducer against deterministic context-free languages is undecidable, where* $X \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RW}, \mathsf{RWW}, \mathsf{RRW}, \mathsf{RRWW}\}$.

Proof. This result is an immediate consequence of the fact that inclusion is not decidable for deterministic context-free languages. Nevertheless we will give a short outline of the argument. Let $(u_1, v_1), (u_2, v_2), ..., (u_n, v_n)$ $(n \geq 1)$ be an instance of the Post Correspondence Problem (PCP) defined on the finite alphabet Σ.[32] With the given instance we associate two deterministic context-free languages $L_1', L_2 \subseteq \Sigma' \cup \{\#\}$, where $\Sigma' = \Sigma \cup \{1, ..., n\}$ and $\Sigma \cap \{1, ..., n\} = \emptyset$. Now

$$L_1' = \{u_{i_1}...u_{i_k}\#i_k...i_1 \mid i_1, ..., i_k \in \{1, ..., n\}\} \text{ and}$$
$$L_2 = \{v_{i_1}...v_{i_k}\#i_k...i_1 \mid i_1, ..., i_k \in \{1, ..., n\}\},$$

for some $k \geq 1$. Obviously the PCP has no solution if and only if $L_1' \subseteq \overline{L_2}$. Clearly $\overline{L_2}$ is computable by a det-mon-R-automaton as these types of automata characterize DCFL and clearly DCFL is closed under complementation.

To establish our result we have to define a language L_1 such that $T(L_1) = L_1'$. Let $T = (Q, \Delta \cup \{\#, \&\}, \Sigma', \Delta, \mathfrak{c}, \$, q_0, 3, \delta)$ be det-mon-R-Td working on the input alphabet

[32] For the undecidability of inclusion as well as for a definition of the PCP we refer to a standard text book, such as [HU79].

$\Delta = \{[u_i] \mid \text{for every pair } (u_i, v_i) \text{ included in the given instance of the PCP}\}$, that is an encoding of the words u_i to single letters $[u_i]$. Now δ is defined by meta-instructions as follows, where $i \in \{1, ..., n\}$:

$$\begin{array}{lll} (\mathfrak{c} \cdot \Delta^*, & \&[u_i]\& \to [u_i]\&; & u_i), \\ (\mathfrak{c} \cdot \Delta^*, & [u_i]\&\# \to [u_i]\#; & u_i\#), \\ (\mathfrak{c} \cdot \Delta^*, & [u_i]\#i \to \#; & i), \\ (\mathfrak{c} \cdot \Delta^* \cdot \# \cdot \$, & \mathsf{Accept}; & \varepsilon). \end{array}$$

Based on that description it is clear that the transition function of T can be designed such that T is monotone and deterministic. Hence, the machine transduces the input language

$$L_1 = \{x\&[u_{i_1}]\&[u_{i_2}]\&...[u_{i_k}]\&\#i_k...i_1 \mid i_1, ..., i_k \in \{1, ..., n\},\ x \in \Delta^*\}$$

such that in the first phase it uses the &-symbol to generate a clone of every $[u_i]$ and in the second phase it checks whether a corresponding sequence of the symbols $[u_i]$ and the indices i are given. Thereby the output language $L_1' = \{u_{i_1}...u_{i_k}\#i_k...i_1 \mid i_1, ..., i_k \in \{1, ..., n\}\}$ is produced. Clearly as the PCP is undecidable and therefore also inclusion for deterministic context-free languages, we achieve the undecidability of type-checking. This completes the proof. □

Chapter 5

Transducing by Observing - A Similar Approach

The basis for the following reflection is the paradigm of Computing by Observing, which is inspired by the way in which experiments are conducted in the natural sciences. It was originally introduced by Cavaliere and Leupold under the name of Evolution and Observation [CL03]. This model somehow breaks with the classical computer science paradigm of processing an input directly into an output, which is the result of the computation. Computing by Observing intends to model the way in which information is gained via experiments. While the actual experiment is running, the results are produced by repeatedly measuring certain quantities like temperature, population size, etc.

Originally introduced for generating [CL04] and accepting formal languages [CL06], the idea of observing and writing a protocol translates very naturally into transductions. Here, we will use its basic structure to define transducers. It consists of an underlying basic system, which evolves in discrete steps from one configuration to the next. An observer reads these configurations and transforms them into output words; a type of classification. This abstracts the protocol of an experiment in biology or chemistry, and for us it is the result of the computation. Figure 5.1 depicts how a sequence of configurations is transformed into a simple sequence of symbols. Obviously one can combine several kinds of formalisms within this architecture. Here we use string rewriting systems as the basic systems and a kind of finite transducer as the observer.

Understandably, the question arises why we introduce these systems in the context of the present work. Besides their unconventional structure, which is worth to investigate

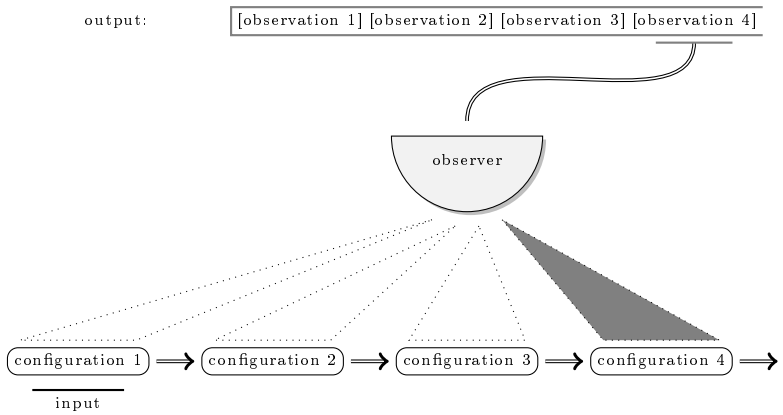

Figure 5.1: Schematic representation of a transducing observer system.

for its own, they offer a different perspective on transductions than exposed in the preceding chapters. Here relations are defined by string rewriting systems that are somehow controlled by a finite state transducer. Restarting transducers can be illuminated in the same way. In contrast, we will see in the following that transducing observer systems offer less control in their computation and more freedom in their definition. Hence, we hope to gain new information on the principles of these mechanisms that use controlled string rewriting systems to realize transductions. In detail, we pursue two aims. First of all, we hope that transducing observer systems somehow form an upper bound for the restarting transductions, for the reason that we can easily gain more power by changing our basic system. Secondly and more importantly, it will turn out that a special type of transducing observer systems seems to be a good candidate for a class of transductions between the ones defined by RRWW- and RWW-transducers. RWW-Td have to restart immediately after a rewrite step and thus cannot read the remainder of the tape. It is an open problem, whether these two classes are equal (see Figure 4.2).

5.1 Definition and Examples

As the present chapter concerns a somehow self-contained topic, we need to define some basic notions, not included in the Preliminaries.

5.1.1 String-Rewriting Systems

The observed systems in our architecture will be string-rewriting systems. Concerning them we follow notations and terminology as exposed by Book and Otto [BO93].

Definition 5.1.1. *A string-rewriting system (*SRS *for short) W on an alphabet Σ is a subset of $\Sigma^* \times \Sigma^*$. Its elements are called rewrite rules, and are written either as ordered pairs (ℓ, r) or as $\ell \to r$ for $\ell, r \in \Sigma^*$.*

The *single-step reduction relation* induced by W is defined for any $u, v \in \Sigma^*$ as $u \Rightarrow_W v$ if and only if there is an $(\ell, r) \in W$ and words $w_1, w_2 \in \Sigma^*$, such that $u = w_1 \ell w_2$ and $v = w_1 r w_2$. The *reduction relation* $\Rightarrow_R^*$ is the reflexive, transitive closure of $\Rightarrow_W$. A string w is called *irreducible* with respect to W, if no rewrite rule from W can be applied to it, i.e. it does not contain any factor that is the left hand side of a rule. The set of all such strings is denoted by $\mathsf{IRR}(W)$.

By imposing restrictions on the set of rewriting rules, many special classes of rewriting systems can be defined. Here we are only interested in the following two special types of rewriting systems.

Definition 5.1.2. *A string-rewriting system is called a length-reducing system if for all its rules (ℓ, r), we have $|\ell| > |r|$, that is, every rule shortens the string by at least one symbol.*

Definition 5.1.3. *A string-rewriting system is called a painter system if for all its rules (ℓ, r), we have $|\ell| = |r| = 1$, that is, every rule just replaces one letter by another one.*

5.1.2 Observers

In the role of observers we use a slightly different version of the devices that have become standard in this function: monadic transducers. They map strings into single letters. Observe that in combination with length-reducing string-rewriting systems this imposes a very strict limit on the right-hand sides of relations. Since every derivation step has to delete at least one symbol, the computed relations are at most *length preserving*. This would make these relations incomparable even to simple classes like the one defined by GSM's; the difference, however could be eliminated by a simple morphism. Therefore, we introduce a slightly modified device, *generalized monadic transducers*, which work just like monadic transducers except for the fact that they output strings. However, we will see that in the observation of painter systems, there is no difference between these two models.

Definition 5.1.4. *A generalized monadic transducer (*gMT *for short) is a tuple* $\mathcal{O} = (Q, \Sigma, \Delta, \delta, q_0, \phi)$*, where the set of states* Q*, the input alphabet* Σ*, the transition function* δ*, and the start state* q_0 *are the same as for deterministic finite automata.* Δ *is the output alphabet, and* ϕ *is the output function, a mapping* $Q \mapsto \Delta^*$ *which assigns an output string or the empty word to each state. The class of all generalized monadic transducers is denoted by* $g\mathcal{MT}$*.*

The mode of operation is that the monadic transducer reads the input word, and then the image under ϕ of the state it stops in, is the output. Note that for every input word there is only one possible output, because the definition is based on deterministic finite automata. Looking at the motivation for the Computing by Observing architecture, this translates as the feature that a given configuration of the underlying system will always be classified in the same way. From a formal point of view this is not necessary, but from the point of view of the motivating examples it seems desirable.

We introduce a few notations that will be convenient in describing the interactions between monadic transducers and string-rewriting systems. For a set of string-rewriting rules W, we will use the notation $W(w_1)$ to denote all sequences of words $(w_1, w_2, \ldots, w_k)$ that form terminating derivations $w_1 \Rightarrow_W w_2 \Rightarrow_W \cdots \Rightarrow_W w_k$ of W. For such a sequence σ and a monadic transducer $\mathcal{O}$, we will denote by $\mathcal{O}(\sigma)$ the result of concatenating all the images of the words in the sequence, that is, $\mathcal{O}(\sigma) = \mathcal{O}(w_1) \cdot \mathcal{O}(w_2) \cdots \mathcal{O}(w_k)$.

5.1.3 Transducing Observer Systems

Now we combine the two components, a string-rewriting system and a monadic transducer in the way described in the introduction.

Definition 5.1.5. *A transducing observer system (*T/O*-system for short) is a triple* $\Omega = (\Sigma, W, \mathcal{O})$*, where* Σ *is the input alphabet,* W *is a string-rewriting system over an alphabet* Γ *such that* $\Sigma \subseteq \Gamma$ *which consists of all the symbols that occur in the rule set* W*, and* $\mathcal{O}$ *is a generalized monadic transducer, whose input alphabet is* Γ*.*

The mode of operation of a transducing observer system $\Omega = (\Sigma, W, \mathcal{O})$ is as follows: the string-rewriting system starts to work on an input word u. After every reduction step the observer reads the new string and produces the corresponding output which is called the observation. The concatenation of all observations of a terminating derivation forms

the output word v. The relation that Ω computes consists of all pairs (u, v) that can be computed in this fashion. Note that already the input string results in the first observation. Thus, there can be an output even if no rewriting rule can be applied to the first string.

Further, the observer is equipped with an important feature: By outputting the special symbol $\perp$ it can abort a computation. In that case no output is produced. The other way in which no output might be produced is that the string-rewriting system does not terminate. Formally, a system like in the definition computes the relation

$$\mathrm{Rel}(\Omega) = \{(u, v) \mid \exists \sigma \in W(u) : \ v = \mathcal{O}(\sigma) \text{ and } |v|_{\perp} = 0\},$$

that is, all pairs formed by an input word and the observations of possible terminating derivations on this input word. Finally, the class of relations defined by a special type X of transducing observing system is denoted by $\mathcal{R}\mathrm{el}(X)$.

5.2 Length-Reducing Systems

We start our investigations by focusing on the relations that are computed by observer systems, where the underlying string rewriting system only uses length-reducing rules (lr-T/O for short). Therefore, it is quiet clear that all pairs of words (u, v) within such a relation fulfill the *length-bounded* property (see Definition 2.1.2), that is, $|v| \leq c \cdot |u|$ for a positive integer c.

Example 5.2.1. Let $\Omega = (\Sigma, W, \mathcal{O})$ be the lr-T/O-system with $\Sigma = \{a\}$, $W = \{aa \to A, AA \to B, BB \to C\}$ and the observer $\mathcal{O}$ that is defined as:

$$\mathcal{O}(w) = \begin{cases} \varepsilon & w \in (a^8)^* \\ a^2 & w \in A^+a^* \\ b^4 & w \in B^+A^* \\ c^8 & w \in C^+B^* \\ \perp & \text{else} \end{cases}.$$

Starting from any word from a^*, only the rule $aa \to A$ can be applied until all a have been consumed from left to right. Every other rule would lead to a string containing B at the same time as a; the observer will output $\perp$ and thus invalidate the transduction. For every two a that are deleted, also two a are output.

In the same way, all A are reduced to B, then all B to C. Every time the number of symbols is divided by two. So for one C there must be two B, four A, and eight a as predecessors. During the computation, eight copies of each a, b, and c are produced. If the original number of a is not divisible by eight, then the observer rejects the input.

Thus, the relation computed by the transducing observer system is

$$R = \mathrm{Rel}(\Omega) = \{(a^n, a^n b^n c^n) \mid n \equiv 0 (\mathsf{mod}\ 8)\}.$$

An immediate consequence of the previous example is the following corollary.

Corollary 5.2.2. *The class of pushdown relations is incomparable to the class of relations computed by* Ir-T/O*-systems.*

Proof. Clearly PDR contains relations that do not fulfill the property that the length of the output word is somehow bounded by the length of the input word. Recall the relation $R = \{(\varepsilon, c^n) \mid n \in \mathbb{N}\}$, which is obviously a pushdown relation but it is not computable by any Ir-T/O-system.

Conversely, in Example 5.2.1 a Ir-T/O-system is described that computes the relation $R = \{(a^n, a^n b^n c^n) \mid n \equiv 0 (\mathsf{mod}\ 8)\}$. This relation is derived by "accepting" the regular language a^n $(n \equiv 0 (\mathsf{mod}\ 8))$ and outputting the context-sensitive language $a^n b^n c^n$. It is well known that pushdown transducers are only capable to map regular languages onto context-free languages (cf. Proposition 2.3.18). □

However, as mentioned in the introduction, we want to relate the transducing observer systems to restarting transducers. Thus, to establish a classification of the power of such systems we first compare them to monotone restarting transducers. In Proposition 4.2.12 it was shown that the class of pushdown functions is included in the class of relations computed by monotone RWW-transducers and monotone RRWW-transducers. In the same spirit we can additionally show the following.

Theorem 5.2.3. PDF $\subsetneq \mathcal{R}\mathrm{el}$(Ir-T/O).

Proof. Obviously the inclusion is proper as the relation shown in Example 5.2.1 is not computable by any functional pushdown transducer. It remains to show that for any pushdown function $R \in$ PDF, there is a Ir-T/O-system Ω such that $R = \mathrm{Rel}(\Omega)$.

Here we again use the technique presented in Theorem 4.2.9. Thus, we construct a pushdown transducer of a very special type by using the notion of simple syntax directed translations scheme (see Definition 2.3.12). However, here we omit further details and proceed directly to the construction of the lr-T/O-system from the mentioned pushdown transducer.

Assume that M is the pushdown transducer presented in the proof of Theorem 4.2.9, which computes a length-bounded pushdown relation. Now M can easily be simulated by a lr-T/O-system $\Omega=(\Sigma, W, \mathcal{O})$. Here the basic idea is that the current state of M and the content of the pushdown store is encoded in the input string of Ω to the left of the current input symbol of M. Without loss of generality the pushdown alphabet is disjoint from the input alphabet. Let $t_1 : (q_1, a, A) \to (q_2, A, v_1)$ and $t_2 : (q_2, b, A) \to (q_3, BC, v_2)$ be two consecutive transitions of M, where q_i is a state, a, b are input symbols, A, B, C are pushdown symbols and v_1, v_2 is the current output. So this transition can be applied in a configuration, where the head of the pushdown transducer is over a letter a in state q_1, and the top of the pushdown store is A. In our representation this corresponds to a string $U[A, q_1, v]abx$, where U is a string of pushdown symbols, v is the output associated to the previous step, and x is a string of input symbols. Then these two steps of the pushdown transducer correspond to one derivation step of our string rewriting system W, that is:

$$U[A, q_1, v]abx \Rightarrow UC[B, q_3, v_1v_2]x.$$

Note that the state is represented in a compound symbol with the last pushdown symbol A next to the current input a.

After this we are back in a configuration equivalent to the one before we started the simulation. Thus, the next transition can be simulated. The length-reducing rules necessary for the simulation are obvious from the derivation. Further, it is clear that a derivation of the above form is only possible if and only if there are corresponding transitions for the pushdown transducer, because all the corresponding symbols must be present in the required positions. The observer is constructed in a way that it executes the following tasks: it has to verify the correct occurrence of pushdown and input symbols within the given string and outputs the output string v_1v_2 given in the compound of the last pushdown symbol. Thus, a generalized monadic transducer $\mathcal{O}$ can control the correct simulation of the transitions by admitting strings of the forms described and by rejecting the computation, if any other type of string appears.

Finally, if the pushdown function contains a pair (ε, v), then we set $\mathcal{O}(\varepsilon) = v$. □

Observe that the latter result does not immediately imply that $\mathcal{R}\text{el}(\mathsf{mon\text{-}RRWW\text{-}Td})$ is properly included in $\mathcal{R}\text{el}(\mathsf{lr\text{-}T/O})$. Monotone restarting transducer compute not only pushdown functions but also all types of pushdown relations that are length bounded (cf. Theorem 4.2.9). This is not true for length-reducing observer systems. For that, consider the fairly simple example relation $\{(\varepsilon, a), (\varepsilon, b), (a, a), (a, b)\}$, which is obviously length bounded and computable by a pushdown transducer. For two reasons, this particular relation is not definable by a $\mathsf{lr\text{-}T/O}$-system. First of all, on empty input such a system is not able to make a non-deterministic choice for the output. As the observers are deterministic devices, possible non-deterministic steps must be realized by the string rewriting system. That means, we simply use auxiliary symbols to express non-deterministic decisions. Recall that a length-reducing observer system consists only of rules that shorten the length of the input string. Hence, we can determine secondly that on an input of length one a $\mathsf{lr\text{-}T/O}$-system is not able to produce more than one output, either. We might overcome this minor issue by introducing non-deterministic observers. However, this would somehow contradict our motivation. Otherwise we strongly expect that we can establish a weaker result, by focusing only on relations where the length of the input is always greater than one, as then the non-determinism can be encoded in the rules of the rewriting system.

Anyway, we omit further discussions on the latter observation, as the corner case mentioned above does not conflict with our basic motivation. So we might assume in the following that we only focus on relations for which every input word has at least length two.

Accordingly, we can show that a restarting transducer that is not monotone is at least as powerful as a length-reducing observer system.

Proposition 5.2.4. $\mathcal{R}\text{el}(\mathsf{lr\text{-}T/O}) \subseteq \mathcal{R}\text{el}(\mathsf{RRWW\text{-}Td})$

Proof. Let $\Omega = (\Sigma, W, \mathcal{O})$ be a $\mathsf{lr\text{-}T/O}$ system, where the observer is $\mathcal{O} = (Q, \Gamma, \Delta, \delta, q_0, \phi)$ and the string rewriting system W is defined as a finite set over $\Gamma^* \times \Gamma^*$. From Ω we build an RRWW-transducer $T = (Q', \Sigma, \Delta, \Gamma', ¢, \$, q_0, k, \delta')$ where k is at least as large as the longest left hand side of a rule $l \to r$ in W and $\Gamma' = \Gamma \cup \{\bar{a} \mid a \in \Gamma\}$.

The main idea of the simulation is that T combines the application of a rule and the observer's behavior. In every cycle T scans the whole tape and determines the output of $\mathcal{O}$ for the current input, that is actually the result of an application of a rule of W to the

tape content of the previous cycle, which belongs to the description of $\mathcal{O}$. Meanwhile non-deterministically T somewhere also shortens the tape content by a rule from W. It is easy to verify that a restarting automaton for the latter behavior is fully described by meta-instructions of the form $(¢ \cdot \Gamma^*, l \mapsto r, \Gamma^* \cdot \$)$ for every rule $l \to r$ in W. Further on, adding the observer to these instructions by applying a standard technique for the intersection of two automata we derive the intended transducer T. Obviously the output of T, when seeing the \$-symbol corresponds to the output function of $\mathcal{O}$, that is $(\mathsf{Restart}, v) \in \delta'(q', \$)$ if and only if $\phi(q) = v$. Here q' denotes a state of T that corresponds to q of the observer. Note that in case $\phi(q) = \bot$, T is simply designed such that it gets stuck, that is, $\delta'(q', \$) = \emptyset$ is taken.

Finally we have to verify accepting conditions for T. Notice that in terms of transducing observer systems, acceptance means that no rule of the string rewriting system is applicable, that is, the current string belongs to $\mathsf{IRR}(W)$. It is well known that $\mathsf{IRR}(W)$ is a regular language. Hence, the set of meta-instructions of T described above simply has to be extended such that also the regular language $\mathsf{IRR}(W)$ is taken into account. Then T recognizes if the current tape content belongs to $\mathsf{IRR}(W)$. In this case it accepts while seeing the \$-symbol and it outputs the corresponding symbols that $\mathcal{O}$ has produced.

The simulation of the transducing observer system Ω by the RRWW-transducer T is quite direct. Thus it is straightforward to see that the same input/output pairs are produced. □

Despite of the reflections outlined before the previous result ("length of the input is always greater than one"), we strongly suspect that the inverse of Proposition 5.2.4 does not hold. This would mean that there are non-trivial relations that can be computed by RRWW-transducers but not by length-reducing T/O-systems. The latter cannot directly connect the output to the rule that has been applied. Typically an intermediate step is used to indicate to the observer, which rule has been applied and where. This trace must then be deleted. If all rules are shortening, only about every second one can be used to produce output in this way, while an RRWW-transducer can rewrite and output (even more than one symbol) in every step. However, we can prove a weakened variant of the inverse inclusion of the one in Proposition 5.2.4. The following result is inspired by the construction of a prefix-rewriting system[33] from an RRWW-automaton, which was shown in [NO00]. Instead of adding a prefix to every rewriting rule we here use a uniform morphism to somehow save

[33] A prefix-rewriting system contains of rewriting rules of the form $xu \to xv$ where x belongs to a regular language.

the information which instruction was applied to the string. In this context a morphism $\varphi : \Sigma^* \to \Gamma^*$ is called uniform if $|\varphi(a)| = |\varphi(b)|$ for every $a, b \in \Sigma$ (e.g. in [RS97], p.339).

Proposition 5.2.5. *For every relation* $R \in \mathcal{R}\mathrm{el}(\mathsf{prop\text{-}RRWW\text{-}Td})$ *there is a uniform morphism* φ *and a relation* $S \in \mathcal{R}\mathrm{el}(\mathsf{Ir\text{-}T/O})$ *such that* $R = \{(u, v) \mid (\varphi(u), v) \in S\}$.

Proof. The basic idea of this proof is to simulate the rewriting steps of a restarting transducer by the rules of a length-reducing system and the prefix (and also suffix) parts, which belong to a regular language, by the observer. For that, the morphism φ transforms words into a redundant representation with a copy of each letter. This allows us to do more than one step in the deletion of a letter by deleting also the copy. The latter is needed to "clean up" after applying a rule. In this way, the Ir-T/O-system can simulate the RRWW-transducer in a very direct way.

Let $T = (Q, \Sigma, \Delta, \Gamma, ¢, \$, q_0, k, \delta)$ be an RRWW-transducer. The morphism is defined as $\varphi(x) = x\hat{x}$ for all $x \in \Gamma$, where the $\hat{x}$ is a copy of the original symbol. Thus, every letter is mapped into two copies. The string-rewriting rules that the T/O-system Ω uses are derived from the meta-instructions of T. Let

$$t : (E_1, u \cdot x \to u', E_2; v)$$

be such a transition for $u, u' \in \Gamma^*$ and $x \in \Gamma$. We associate to each transition a unique label, here t. From this description we build a Ir-T/O-system $\Omega = (\Sigma, W, \mathcal{O})$ as follows, where $W \subseteq \Gamma'^* \times \Gamma'^*$ and Γ' consists of $\Gamma \cup \{\hat{x} \mid x \in \Gamma\}$ as well as some special symbols described below. The string-rewriting rule, which is added to W, for the meta-instruction t is $\varphi(ux) \to \varphi(u') \cdot t$. Thus, the additional space induced by φ is used to assign the corresponding label to the applied instruction.

Further we add $t \to \varepsilon$. While $x\hat{x}$ is deleted, the observer produces the string that T outputs in the execution of t. As described in the meta-instruction, this string is v. For every meta-instruction of T the observer's mapping includes the clause

$$\mathcal{O}(w) = v; \text{ if } w \in \varphi(E_1) \cdot \varphi(u') \cdot t \cdot \varphi(E_2).$$

In this clause, we check whether t was really applicable. This means that T can reach the state in which the rewrite operation of t is applied after reading the prefix of the current string until the application site of the rule. Of course, the part ux is not there anymore, when this clause is applied. But since there is the symbol t, a monadic transducer can

recognize this symbol and act as if ux was there, that is, the current string must belong to the language $\varphi(E_1) \cdot \varphi(u') \cdot t \cdot \varphi(E_2)$ rather than to $E_1 \cdot u \cdot E_2$, which are the strings that allow application of the meta-transition t. This is still a regular condition to check. Note that all the clauses, which describe the observer, are disjoint, due to the presence of t. Thus, the observer always produces the desired output. Additionally after applying a meta-instruction the label t has to be deleted by the rule $t \to \varepsilon$ and in this case the observer has a clause of the form

$$\mathcal{O}(w) = \varepsilon; \text{ if } w \in \varphi(\Gamma^*).$$

In an analogous way we treat accepting cycles. In the case that a computation of T accepts with empty tape such that the observer system has also no symbols left, then clearly, Ω stops as well, with the same output as T. The second case is more problematic, where T accepts and Ω has still symbols left. As long as there are symbols left, we can rewrite any of them to a symbol of an accepting transition t_a as above.

Observe that Ω will not stop, when the symbol t_a is introduced. But the computation of a T/O-system is complete only when no more rule can be applied. That is why we use the special symbol t_a to delete every remaining symbol, that is, we add rules $xt_a \to t_a$ and $t_a x \to t_a$ for all symbols x in the alphabet of W. That is, t_a absorbs all the symbols that are left. With the presence of t_a the observer maps any further string to the empty output. So when there is only t_a left, the system stops and has output the same string as T.

Finally, we explain how Ω behaves on rejecting computations of T. Note, that T rejects an input word simply by getting stuck, that is, no transition is applicable in the current configuration. As the clauses of the observer mirror directly the move-right steps of T, it will also get stuck. In this situation we have to output $\perp$ to abort the computation of Ω. This can be done by making the observer "complete", that is, the transition function of $\mathcal{O}$ is extended such that every input word w, which is not described by the regular expressions above leads to the following output:

$$\mathcal{O}(w) = \perp.$$

This completes the proof. □

Of course, we would like to be able to do the simulation without any morphism and thus show the equivalence of the two models. But with the techniques used here this seems not

to be possible, as briefly shown by the latter proof. The introduction of the symbol t is necessary to signal to the observer which instruction is simulated and what needs to be output, then another step is needed to clean up this trace. But in a length-reducing system, every step consumes at least one symbol. Thus, we could only simulate RRWW-transducers whose rules shorten the string at least by two in every step.

5.3 Painter Systems

As mentioned in the introduction, by adjusting the rules of transducing observer systems we are able to establish an upper bound for the relations computed by restarting transducers. To this end, we now concentrate on painter systems (pnt-T/O for short). Recall that a string rewriting system that uses painter rules is only allowed to rename symbols, that is, for every rule $l \to r$, the length bound $|l| = |r| = 1$ holds. Note again that we might run into the same problem as exposed in the previous section. On empty input the observer is not able to produce several outputs. As we do not want to change our definitions for now, we avoid this very special case by focusing on proper restarting transducers.

Proposition 5.3.1. $\mathcal{R}\text{el}(\text{prop-RRWW-Td}) \subseteq \mathcal{R}\text{el}(\text{pnt-T/O})$

Proof. Recall that we stated a similar result in Proposition 5.2.5. There we applied a uniform morphism on the input to gain some space. This additional space was needed to save the trace of which meta-instruction was applied. Here, as the rules of the current observer system are not length reducing, all tape cells can be rewritten any number of times. Hence, there is no need for additional space. Thus, we can save the trace directly together with the current input string.

Without loss of generality let $T = (Q, \Sigma, \Delta, \Gamma, ¢, \$, q_0, k, \delta)$ be a proper RRWW-Td that restarts and accepts only on the \$-symbol. Further, assume that T is described by labeled meta-instructions, where each rewriting meta-instructions of T is defined as follows:

$$t : (E_1, u_1u_2...u_k \to u'_1u'_2...u'_k, E_2; v),$$

where $u_i \in \Gamma$ and $u'_i \in \Gamma \cup \{\varepsilon\}$ $(i \in \{1, \ldots, k\})$ is the corresponding symbol that occurs in the string produced by the current meta-instruction. Note that we here use a slightly different representation of meta-instructions, that is, each rewriting rule $u \to u'$ is described as a unique letter to letter mapping, where at least one letter is mapped to the empty word.

For instance, let $u = u_1u_2...u_k$ and $u' = u'_1u'_2...u'_k$, where $u_i, u'_i \in \Gamma \cup \{\varepsilon\}$ $(i \in \{1, ..., k\})$, then the rule $u \to u'$ can be depicted as $u_1u_2...u_k \to u'_1u'_2...u'_k$, which implies that u_1 is rewritten to u'_1 and so on. Further, each accepting meta-instruction is also uniquely labeled by $t_a : (E, \mathsf{Accept}; \varepsilon)$. From the description of T we build a pnt-T/O-system $\Omega = (\Sigma, W, \mathcal{O})$, where the observer is defined as $\mathcal{O} = (Q', \Gamma' \cup \{\not\vdash, \$\}, \Delta, \delta', p_0, \phi)$ and we consider the string rewriting system W as a set of rules over $(\Gamma' \cup \{\not\vdash, \$\})^* \times (\Gamma' \cup \{\not\vdash, \$\})^*$, where Γ' consists of Γ, the set $\{a^t, a^{tt}, a^{t_a} \mid a \in \Gamma \cup \{\not\vdash, \$\}$ and t, t_a are labels of rewriting or accepting meta -instructions of $T\}$, the special auxiliary symbols λ and f, which denote erasing rules, and the set $\{\lambda^a, a^\lambda \mid a \in \Gamma \cup \{\$\}\}$, which is needed to "remove" erased symbols. In the following, the latter set is needed to simulate accepting meta-instructions by the observer system. Additionally we assume that every input string of W is also bounded by the markers $\not\vdash$ and $\$$. At the end of the proof we will show that this is not a necessary assumption. Anyway it increases the readability of the technical details.

Here we show for one concrete case how the set of rewriting rules W is defined, and how the observer is used to control the derivation of W.

For each meta-instruction $t : (E_1, u_1u_2...u_k \to u'_1u'_2...u'_k, E_2; v)$ of T , we add the rules $u_i \to u_i^t$, $u_i^t \to u_i^{tt}$ and $u_i^{tt} \to u'_i$ $(i \in \{1, ..., n\})$ to W. If $u'_i = \varepsilon$, then we add the rule $u_i^{tt} \to \lambda$, where λ is an auxiliary symbol. Now the observer's mapping for an input $w \in (\Gamma \cup \{\not\vdash, \$\})^*$ includes the clauses

$$\mathcal{O}(w) = \begin{cases} \varepsilon; & \text{if } w \in E_1 \cdot u_1u_2 \cdots u_k \cdot E_2, \\ \varepsilon; & \text{if } w \in E_1 \cdot u_1^t u_2 \cdots u_k \cdot E_2, \\ \vdots & \vdots \\ \varepsilon; & \text{if } w \in E_1 \cdot u_1^t u_2^t \cdots u_k^t \cdot E_2, \\ \varepsilon; & \text{if } w \in E_1 \cdot u_1^{tt} u_2^t \cdots u_k^t \cdot E_2, \\ \vdots & \vdots \\ v; & \text{if } w \in E_1 \cdot u_1^{tt} u_2^{tt} \cdots u_k^{tt} \cdot E_2, \\ \varepsilon; & \text{if } w \in E_1 \cdot u_1^{'} u_2^{tt} \cdots u_k^{tt} \cdot E_2, \\ \vdots & \vdots \\ \varepsilon; & \text{if } w \in E_1 \cdot u_1^{'} u_2^{'} \cdots u_k^{'} \cdot E_2. \end{cases}$$

Let $t : (E_1, u_1u_2...u_i \ldots u_k \to u'_1u'_2...u'_i \ldots u'_k, E_2; v)$ be a meta-instruction that is applicable to a restarting configuration $(q_0 \not\vdash xu_1u_2 \cdots u_k y\$, v')$, where $\not\vdash \cdot x \in E_1$, $y \cdot \$ \in E_2$, $u'_i = \varepsilon$,

and $v' \in \Delta^*$. Hence, this leads to a cycle of the form:

$$(q_0 \cent x u_1 u_2 \dots u_k y \$, v') \vdash_T^c (q_0 \cent x u'_1 u'_2 \dots u'_k y \$, v'v).$$

Then the transitions of T are simulated by Ω as follows:

$$\begin{array}{rlcll} & \cent x u_1 u_2 \dots u_i \dots u_k y \$ & \Rightarrow & \cent x u_1^t u_2 \dots u_i \dots u_k y \$ & \Rightarrow \\ \dots \Rightarrow & \cent x u_1^t u_2^t \dots u_i^t \dots u_k^t y \$ & \Rightarrow & \cent x u_1^{tt} u_2^t \dots u_i^t \dots u_k^t y \$ & \Rightarrow \\ \dots \Rightarrow & \cent x u_1^{tt} u_2^{tt} \dots u_i^{tt} \dots u_k^{tt} y \$ & \Rightarrow & \cent x u'_1 u_2^{tt} \dots u_i^{tt} \dots u_k^{tt} y \$ & \Rightarrow \\ \dots \Rightarrow & \cent x u'_1 u'_2 \dots \lambda \dots u'_k y \$ & & & \end{array}.$$

After this we are back in a situation similar to the one before we started the simulation. The only difference is that after the first simulation of a meta-instruction, the configuration of W is interspersed with the λ-symbol. These special symbols have to be "removed", as otherwise possible rewriting rules of W, which are taken from the description of the restarting transducer, might not be applicable. Recall that a painter system is not allowed to delete any symbol. Hence, we need to introduce additional rules for W and clauses for the observer such that every occuring λ is shifted to the right of the \$-symbol before the next rule of T is applied. This can be done by applying a technique exposed in [CL06] to simulate context-sensitive rules in the form of $AB \to BA$ by painter systems. Hence, without going into details we can obviously add rules to W such that the observer enables the following derivation

$$\lambda a \Rightarrow_W \lambda^a a \Rightarrow_W \lambda^a a^\lambda \Rightarrow_W a a^\lambda \Rightarrow_W a\lambda,$$

for every $a \in \Gamma \cup \{\$\}$. Clearly every clause, which has to be added to the observer to enable the previous derivation, leads to an empty output. Further, the special λ-symbols right of the \$-symbol must be ignored by the observer, that is, it reads over them without a change of state. Thus, the next transition can be simulated. Further notice, that all the intermediate configurations can be described by disjoint regular expressions that are specific to the transition t, or the symbols λ^a, a^λ, because they contain some versions of these symbols.

By exhaustive checking, we can verify that application of these rules in any other order, or the application of a rule stemming from another transition will lead to a string not described by these expressions. Therefore, it is shown that a generalized monadic transducer can control the correct simulation of these transitions by admitting strings of the forms

described and by rejecting the computation and outputting $\perp$, if any other type of string appears.

Further, for each accepting meta-instruction $t_a : (E, \mathsf{Accept}; \varepsilon)$, we add the rules $a \to a^{t_a}$ and $a^{t_a} \to f$ to W, where $a \in \Gamma \cup \{\text{¢}, \$\}$. Then the observer only has to check whether t_a occurs in its input and whether the input corresponds to the regular language E. The special symbol f is again needed to force the system to stop, that is, when f is introduced, there are no rules to "delete" f. Thus, after f occurs in a configuration, there possibly is a finite number of additional rewrite steps. Hence, the observer is defined such that all configurations where the symbol f is included lead to the empty output.

Finally, by adding additional symbols and rewriting rules to Ω that mark the first and last letter of the input string at the beginning of the computation, we can omit the assumption that every input string contains the special symbols ¢ and \$. Clearly, then every rule in W and the accepting conditions have to be adjusted. So, it is shown that for every proper RRWW-transducer T, there is a pnt-T/O-system Ω such that $\mathrm{Rel}(T) = \mathrm{Rel}(\Omega)$. □

Actually, the previous inclusion is proper, as it is clear that the relation $\{(a, a^n) \mid n \geq 1\}$ is computable by the pnt-T/O-system $\Omega = (\Sigma, W, \mathcal{O})$, where $W = \{a \to a, a \to f\}$ and

$$\mathcal{O}(w) = \begin{cases} a; & \text{if } w = a, \\ \varepsilon; & \text{if } w = f, \\ \perp; & \text{if } w \neq a \text{ and } w \neq f. \end{cases}$$

Note that this particular relation violates the length-bounded property. Thus, the next corollary holds.

Corollary 5.3.2. $\mathcal{R}\mathrm{el}(\mathsf{prop\text{-}RRWW\text{-}Td}) \subsetneq \mathcal{R}\mathrm{el}(\mathsf{pnt\text{-}T/O})$

The latter example shows that relations computed by painter systems are not necessarily length bounded. Hence, in contrast to restarting transducers we can show that nearly every rational relation is computable by such a system.

Theorem 5.3.3. *Each rational relation $R \subseteq \Sigma^* \times \Delta^*$ that contains only one pair in the form of (ε, v) ($v \in \Delta^*$) is computable by a* pnt-T/O*-system $\Omega = (\Sigma, W, \mathcal{O})$.*

Proof. This result seems quite obvious. Nevertheless, it is not an immediate consequence of the results presented in this Chapter. By Proposition 5.3.1 and by the fact that mono-

tone RRWW-transducer characterize the length-bounded pushdown relations (cf. Theorem 4.2.9) it is clear that there is a pnt-T/O-system for every relation $R \in$ lbPDR, where $(\varepsilon, v) \notin R$ for any word v over the output alphabet. Further, as lbPDR is obviously a superclass of the rational relations for which the length-bounded property holds, we only have to show how a pnt-T/O system works on rational relations that are not length bounded. Note that the violation of this property is caused, for instance, by finite state transducers that produce non-empty output, when performing cycles without reading any symbol.

To begin with, we may assume that $\Omega = (\Sigma, W, \mathcal{O})$ is a pnt-T/O-system that simulates a finite state transducer $T = (Q, \Sigma, \Delta, \delta, q_0, F)$ that computes a length-bounded rational relation. We further assume that Ω is designed similar to the lr-T/O system that simulates a pushdown transducer shown in the proof of Theorem 5.2.3. Thus, when T performs a step $(qabu, v) \vdash_T (q'bu, v\alpha)$ with a transition in the form of $t : \delta(q, a) = (p, \alpha)$, where $q, p \in Q$, $a, b \in \Sigma$, $u \in \Sigma^*$ and $v, \alpha \in \Delta^*$, then Ω simulates this step by the following derivation,

$$\lambda^*[q,a]bu \Rightarrow_W \lambda^*[q,a,p,\alpha]_t bu \Rightarrow_W \lambda^*[q,a,p,\alpha]_t b^t u \Rightarrow_W \lambda^*[q,a,p,\alpha]_t[p,b]u \Rightarrow_W \lambda^*\lambda[p,b]u,$$

where λ, $[q,a]$, $[q,a,p,\alpha]_t$, b^t and $[p,b]$ are auxiliary symbols not in Σ. Admittedly, this representation might be quite redundant, but it is clear from the derivation, how the rules in W have to be obtained from the transitions of T. Further, the different configurations in a derivation can obviously be described by disjoint regular expressions, which is the definition of the observer $\mathcal{O}$. Additionally, the observer outputs α when scanning the configuration $\lambda^*[q,a,p,\alpha]_t bu$. Up to now, this simulation is simply a consequence of previous results, as stated above.

We next describe how Ω mirrors possible ε-steps of a finite state transducer, which causes the violation of the length-bounded property. For that we add a transition in the form of $t_\varepsilon : \delta(q, \varepsilon) = (q', \beta)$ to the transition function of T, where q is the state used above, $q' \in Q$ and $\beta \in \Delta^*$. Thus, we have to extend the string rewriting systems by rules that are in some sense derived by calculating the ε-closure[34] of a state of T. Here, for q we have to add the rules $[q,a] \to [q,a,p,\alpha]_t$, $[q,a] \to [q,a,q',\beta]_{t_\varepsilon}$ and, $[q,a,q',\beta]_{t_\varepsilon} \to [q',a]$, which mirrors the returning to the actual configuration of the finite state transducer. Clearly, by introducing the special symbol t_ε the observer can be adjusted, such that it enables

[34] The ε-closure of a state q is the set of states that are reachable from q by zero or more ε-transitions (e.g. in [RS97], p.52).

derivations of W in the form of

$$\lambda^*[q,a]bu \Rightarrow_W \lambda^*[q,a,q',\beta]_{t_\varepsilon}bu \Rightarrow_W \lambda^*[q',a]bu,$$

and that it outputs β when reading $\lambda^*[q,a,q',\beta]_{t_\varepsilon}bu$. Further, observe that the example transition t_ε already covers all cases of ε-transitions occuring in a description of a finite state transducer. Finally, if the rational relation R contains one pair (ε, v) $(v \in \Delta^*)$ we simply add the clause $\mathcal{O}(\varepsilon) = v$ to the observer. For that note, a finite state transducer that computes such a relation R is not able to perform a cycle of ε-steps on empty input, as this would violate the property that only one pair is in the form of (ε, v). Hence, v can easily be obtained by calculating a kind of ε-closure for the initial state q_0 of T. Thus, it is shown that for every relation $R \in \mathsf{RAT}$, which is restricted in the above way, there is a pnt-T/O-system Ω such that $R = \mathrm{Rel}(\Omega)$. □

The previous result cannot be adapted to compute pushdown relations by pnt-T/O-systems.

Proposition 5.3.4. *The class of pushdown relations is incomparable to* $\mathcal{R}\mathrm{el}$(pnt-T/O).

Proof. Clearly by Proposition 5.3.1 there are relations computed by pnt-T/O that are not pushdown relations. Conversely, besides the trivial example of producing several outputs on the empty input, consider the relation $R = \{(c, a^n b^n) \mid n \geq 0\}$. Obviously R is a pushdown relation. A pushdown transducer for R uses ε-steps to push a number of symbols on the stack while outputting the same number of a's. At some point of the computation it decides non-deterministically to pop all stack symbols while again outputting the same number of b's. Finally it just has to check that there is only one c on the tape.

A pnt-T/O-system for R gets c as input string. Hence, it must rewrite c to produce an output. As there is only a finite number of rewriting rules, c has to be rewritten in a form of cycle-rules to produce an arbitrary number of a's. Thus, it is obvious that there is no possibility to save the number of rules that were used to produce a's. Therefore, the pnt-T/O-system is not able to output the same number of b's. Hence, $R \notin \mathcal{R}\mathrm{el}$(pnt-T/O). □

On the other hand the mode of operation of painter systems suggests that possibly a class of relations that is definable by linear bounded automata could be simulated by these systems. Changing a state and rewriting adjacent cells in the string in a coordinated manner could suffice for this. However, in the literature little can be found on relations

computed by linear bounded automata[35], and furthermore, there is no uniform theory on these relations. That is maybe because starting from the context-sensitive languages, the distinction of transductions and plain sets is not so significant anymore.

We conclude by stressing that, from a theoretical point of view, this chapter was meant to be a starting point for gaining a different perspective on relations that can be somehow associated to string rewriting systems. For that observe again that it seems to be appropriate to disregard the original motivation of observer systems and change our definitions such that the corner case (output on empty input) is avoided. However, this does not solve our main problem: Is there a non-trivial witness relation between lr-T/O and RRWW-Td? Such a relation might also be a guide to find a relation between RWW-Td and RRWW-Td.

[35] An exception is the paper of Ginsburg and Rose [GR66], which indeed implies that there might be a connection between relations computed by LBA and pnt-T/O-systems, with respect to the results established in the present section.

Chapter 6

Conclusion

We conclude by giving some thoughts on the results of this thesis. For that, we first discuss theoretical results and then turn to some reflections on possible applications. We end our discussion by posing a list of questions left open, which are attractive from the author's subjective point of view.

Even though the results on restarting transducers presented in the preceding chapters are far from complete, they have shown that extending restarting automata to transducers proved to be "natural" in the sense that there are restarting transducer characterizations for several traditional relation classes. Mainly, we may summarize that the behavior of restarting transducers is comparable to well-known types of transducers, which are not capable of performing ε-steps. In that context relations computed by restarting transducers with window size one have to be emphasized. These models gave a new insight into the well-known hierarchy of subclasses of rational relations. By investigating those types of restarting transducers where the underlying automata characterize the regular languages, we were able to show how the decrease or increase of power imposed by the various restrictions and extensions (i.e. mon-, det-, nf-) leads to equivalences to the rational functions (RATF), general sequential machine relations (GSMRel), subsequential functions (SubSeqF), and deterministic general sequential machine functions (dGSMF). This offers a different perspective on both, the capabilities needed to compute instances of these classes, and on the power gained by the above restarting automata restrictions or extensions.

According to the Introduction, the results on restarting transducers further imply that these machines are a promising tool for establishing a complete framework for transductions based on this singular model only. In general, we have defined a hierarchy of length-

bounded relations along the traditional notions of pushdown relations (PDR) and rational relations (RAT). Furthermore, we derived some new relation classes above the length-bounded pushdown relations (lbPDR), characterized uniquely by restarting transducers.

In retrospective, Chapter 3 ("Relations Computed by Restarting Automata and Parallel Communicating Systems") and Chapter 5 ("Transducing by Observing") play some kind of supporting role for results on restarting transducers. In particular, we mainly used these chapters to establish some upper bounds for the computational power of different types of restarting transducers. Nevertheless, the additional results obtained show already that realizing transductions by the notion of input/output- and proper relations, PC-systems of restarting automata and by the principle of observing a string rewriting system is reasonable for several reasons. Fruitful tasks for these mechanisms seem to be, for instance, defining new classes of relations in the spirit of Aho and Ullman's notion of characterizing languages [AU69] or gaining deeper insights into traditional relation classes itself by representing them from an unconventional perspective.

Finally, we want to outline a few thoughts on possible applications in the field of linguistics of the previously introduced models. In general, it is well known that models for realizing subclasses of rational relations play an important role in natural language processing and speech recognition [KK94, Moh97, JM09][36]. Hence, restarting transducers of the type investigated in Subsection 4.3 might also fit these tasks. In particular, several of the authors mentioned above described usage scenarios of types of finite state transducers in the process of morphological analyzation, that is, roughly speaking, realizing relations between surface and lexical forms of words. Hence, for instance subsequential transducers are used to offer succinct representations of morphological dictionaries [Moh97]. As mentioned in the preliminaries it is known from Kutrib and Reimann [KR08, Rei07] that in terms of descriptional complexity, there is a benefit in using (forgetting) restarting automata to represent regular languages. Furthermore, we have shown in Proposition 2.2.15 that non-forgetting restarting automata yield even more succinct representations than (forgetting) restarting automata. Obviously, this effect carries over to the corresponding classes of transducers by extending the witness languages to relations. Therefore, restarting transducers with window size one (especially det-mon-nf-R(1)-Td) might be a reasonable alternative for certain finite state transducer applications in natural language processing, such as realizing morphological dictionaries.

[36] Admittedly, as applications of transducers play a minor role in this work, the selected citations are only meant to be a starting point for further reading.

Another possible application directly concerns the original motivation of restarting automata. We already mentioned in the introduction that one of our main motivations for extending this model to a transducing device was to extract information on the given input sentence during the process of analyzation. In Section 2.2 we briefly pointed out that Analysis by Reduction applied on a sentence of a natural language additionally provides morphological and dependency information on the input. Hence, presenting this information in a certain form might be a suitable task for a restarting transducer. In recent years this latter scenario was substantiated by Lopatkova et al. [LPS07, LMP10] and Plátek et al. [PML10b, PML10a]. They considered restarting automata that serve as a formal model for the Functional Generative Description (FGD), that is, a dependancy based descriptive system based on the notion of Analysis by Reduction. This framework includes a four step analyzation of sentences in Czech. In the first three steps a given input is annotated with linguistic categories such as morphological and syntactical informations, with the aim of disambiguation. Then this (at least in principle) disambiguated sentence is translated by Analysis by Reduction into a tectogrammatical representation, that is, a dependency tree (so to say, a kind of semantic structure). The above authors have shown that restarting automata are capable of mirroring this enriched form of Analysis by Reduction (applied before the fourth step), where in [PML10b, PML10a, LMP10] a type of restarting automaton is suggested which additionally outputs a dependency tree like structure. Hence, general investigations on restarting automata with output (i.e. restarting transducers) are mandatory.

Open Problems

Finally, we present a list of questions left open throughout this work as well as some suggestions on resolving them. The order of the different problems stated here mirrors the grade of attraction to the author.

- *Is the inclusion* $\mathcal{R}\text{el}(\mathsf{det\text{-}RWW\text{-}Td}) \subseteq \mathcal{R}\text{el}(\mathsf{det\text{-}RRWW\text{-}Td})$ *presented in Figure 4.2 proper?* It is well known that this question has to be answered negatively for the corresponding types of automata. However, the proof does not carry over to transducers. The equivalence $\mathcal{L}(\mathsf{det\text{-}RWW}) = \mathcal{L}(\mathsf{det\text{-}RRWW})$ of the language classes is a direct consequence of the fact that both types of automata characterize the Church-Rosser languages. A summary of this topic can be found in [Nie02]. There the author mainly showed that the classical model for the Church-Rosser languages,

the shrinking deterministic two pushdown automaton (sDTPDA for short), coincides with its length-reducing counterpart (lrDTPDA for short). Additionally a det-RWW-automaton can simulate a lrDTPDA, and a det-RRWW-automaton can be simulated by a sDTPDA.

For our purposes it seems to be inappropriate to extend these proofs in terms of transducers, as it is not clear whether for a transducer (restarting or pushdown) being shrinking instead of being length-reducing does not lead to more complicated transductions. For that note that a shrinking restarting transducer can be obtained from the definition of the shrinking restarting automaton, that is, such a device has the additional ability to rewrite according to a weight function (see [JO07]). Recall that "normal" restarting transducers are only able to apply length-reducing rewrite steps. In fact, we suspect that shrinking is more powerful than length-reducing in terms of deterministic RRWW-transducers. This guess is substantiated by the following simple counting argument. On a given input of length n, a shrinking restarting transducer of this type is able to perform $k \cdot n$ many restarts, where k depends on the used weight function. Clearly, non-shrinking transducers can only perform n restarts. If it is possible to encode the additional number of restarts in a meaningful output, the so-defined relation classes do not coincide.

Furthermore, returning to the actual question, we might show the equivalence directly. Thus, a det-RWW-Td has to be constructed that simulates a det-RRWW-Td. Here the method of choice would be to force the det-RWW-Td to collect all the information that can occur in a right-computation[37] of a det-RRWW-Td and verify it within the tail of its computation. This works for automata, as it is well known that these types of machines are weakly monotone. A proof which uses the above technique can be found for instance in [Sch10]. Unfortunately, this technique does not carry over to restarting transducers, as it is not possible for a det-RWW-Td to collect additionally all the outputs produced during all the right-computations of a det-RRWW-Td and outputting them in the tail of its own computation. Hence, a more involved technique seems to be needed, such as a preprocessing of the input from right to left by using some compression arguments in order to collect all informations on possible right-computations first. However, after presenting a sketch of the thoughts spent on this question, we are still not quite sure about the answer.

[37] Here a right-computation denotes the part of one cycle of a RR(W)(W)-machine that follows after rewriting.

- *Is there a (non-trivial) witness relation between* $\mathcal{R}\text{el}(\text{lr-T/O})$ *and* $\mathcal{R}\text{el}(\text{RRWW-Td})$*?* Recall from Chapter 5 that there is a trivial relation R with $R \notin \mathcal{R}\text{el}(\text{lr-T/O})$ and $R \in \mathcal{R}\text{el}(\text{RRWW-Td})$, where simply the empty word is mapped to several non-empty outputs. Nevertheless, when we restrict the search for a witness to relations, where the input is at least of length two, the problem becomes difficult. Although we strongly suspect that both classes can be non-trivially separated, we do not even have a candidate. We assume that finding such a witness might be promising for another open question; *do the classes* $\mathcal{R}\text{el}(\text{RRWW-Td})$ *and* $\mathcal{R}\text{el}(\text{RWW-Td})$ *coincide or not?* The reason is that a technique that proves a relation to be in $\mathcal{R}\text{el}(\text{RRWW-Td})$ and not to be in $\mathcal{R}\text{el}(\text{lr-T/O})$ might be a useful framework in the general context of relations defined by regular language controlled string rewriting systems. Furthermore, observe that $\mathcal{R}\text{el}(\text{RRWW-Td}) \stackrel{?}{=} \mathcal{R}\text{el}(\text{RWW-Td})$ is an instance of the longstanding open question $\mathcal{L}(\text{RRWW}) \stackrel{?}{=} \mathcal{L}(\text{RWW})$ for the corresponding types of restarting automata (e.g. posed in [JMPV98]). Therefore, getting some new arguments for the first question might also bring us closer to answering $\mathcal{L}(\text{RRWW}) \stackrel{?}{=} \mathcal{L}(\text{RWW})$.

- This point generally concerns closure properties and decision problems of restarting transducers, as they are only sparsely studied in the present thesis. In particular, except for prop-det-R(1)-transducers that output only single symbols (cf. Corollary 4.4.9), we have shown in Section 4.4 that all other types of restarting transducers considered in this work are not closed under composition. As mentioned before, composition is a crucial property in certain applications. Hence, *are there any types of restarting transducers that are closed under composition?* Furthermore, the composition operation itself might help to put certain types of restarting transducers, which are not taken into account yet, in relation to well-known classes of transductions. For instance, we suspect that general non-forgetting restarting transducer are closed under composition with some length-bounded subclasses of the rational relations. A proof for this seems quite obvious; a finite state transducer is simply simulated in the finite control of the non-forgetting machine. Hence, according to results on composing pushdown functions with certain types of rational relations, which were presented by Choffrut and Culik II [CI83], such investigations seem valuable to derive further classification results for restarting transducers.

  Up to now, we only know little about relation classes between dGSMF and DPDF, which are defined by restarting transducers. According to Section 2.3 equivalence is decidable for the class of deterministic pushdown functions (see [Sén99]). Hence,

are there further relation classes that are defined by restarting transducers, for which equivalence is decidable?

- A motivation of this work is linguistics. Further, we have shown that from a theoretical point of view restarting transducers with window size one offer some suitable properties for applications in natural language processing. Thus, *is there a reasonable example from linguistic applications that proves the worthiness of restarting transducers with window size one in this context?*

- Last but not least, we have seen in Subsection 4.3.3 that $\mathsf{R}(1)$-transducers are able to map regular languages to context-free ones. Moreover, we assume that this result even extends to context-sensitive languages. This seems surprising, at first sight, as it is well known that $\mathsf{R}(1)$-automata characterize the regular languages. Obviously, the mentioned results are based on the fact that these types of transducers are able to delete symbols "anywhere" from the tape. This can be interpreted as the possibility of moving the head freely. Therefore, the comparison of $\mathsf{R}(1)$-transducers to *two-way finite state transducers* (e.g. exposed in [EY71]) seems to be promising, as both machines show a similar behavior in terms of their non-preservation of language classes.

Of course, the previous list is only a selection of questions left open throughout this work. Additionally, besides the restrictions and extensions of restarting automata considered in the present thesis, this research area includes a variety of conventional and unconventional mechanisms based on this model, such as two-way restarting automata [Plá01], restarting tree automata [Sta08], or cooperating distributed systems of (simple) restarting automata [Mes08, NO12]. From a linguistic point of view the extension of restarting tree automata to transducers, for instance, might be a good starting point for further research.

Bibliography

[AHU69] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. A General Theory of Translation. *Mathematical Systems Theory*, 3(3):193–221, 1969.

[AU69] Alfred V. Aho and Jeffrey D. Ullman. Properties of Syntax Directed Translations. *J. Comput. Syst. Sci.*, 3(3):319–334, 1969.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[Ber79] Jean Berstel. *Transductions and Context-Free Languages*. Leitfäden der angewandten Mathematik und Mechanik. Teubner, 1979.

[BH77] Meera Blattner and Tom Head. Single-Valued a-Transducers. *Journal of Computer and System Sciences*, 15(3):310 – 327, 1977.

[BO93] Ronald V. Book and Friedrich Otto. *String-Rewriting Systems*. Texts and monographs in computer science. Springer, 1993.

[Cho77] Christian Choffrut. Une Caracterisation des Fonctions Sequentielles et des Fonctions Sous-Sequentielles en tant que Relations Rationnelles. *Theor. Comput. Sci.*, 5(3):325–337, 1977.

[CI83] Christian Choffrut and Karel Culik II. Properties of Finite and Pushdown Transducers. *SIAM J. Comput.*, 12(2):300–315, 1983.

[CL03] Matteo Cavaliere and Peter Leupold. Evolution and Observation: A New Way to Look at Membrane Systems. In Carlos Martín-Vide, Giancarlo Mauri, Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Workshop on Membrane Computing*, volume 2933 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2003.

[CL04] Matteo Cavaliere and Peter Leupold. Evolution and Observation — A Non-Standard Way to Generate Formal Languages. *Theoretical Computer Science*, 321:233–248, 2004.

[CL06] Matteo Cavaliere and Peter Leupold. Observation of String-Rewriting Systems. *Fundam. Inform.*, 74(4):447–462, 2006.

[DW86] Elias Dahlhaus and Manfred K. Warmuth. Membership for Growing Context-Sensitive Grammars is Polynomial. *J. Comput. Syst. Sci.*, 33(3):456–472, 1986.

[Eil74] Samuel Eilenberg. *Automata, Languages, and Machines*, volume A. Academic Press, Inc., Orlando, FL, USA, 1974.

[EM65] Calvin C. Elgot and Jorge E. Mezei. On Relations Defined by Generalized Finite Automata. *IBM J. Res. Dev.*, 9(1):47–68, January 1965.

[EY71] Roger W. Ehrich and S. S. Yau. Two-Way Sequential Transductions and Stack Automata. *Information and Control*, 18(5):404–446, 1971.

[FRR+10] Emmanuel Filiot, Jean-François Raskin, Pierre-Alain Reynier, Frédéric Servais, and Jean-Marc Talbot. Properties of Visibly Pushdown Transducers. In *MFCS*, pages 355–367, 2010.

[GR66] Seymour Ginsburg and Gene F. Rose. Preservation of Languages by Transducers. *Information and Control*, 9(2):153–176, 1966.

[GR68] Seymour Ginsburg and Gene F. Rose. A Note on Preservation of Languages by Transducers. *Information and Control*, 12(5/6):549–552, 1968.

[Har78] Michael. A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1978.

[HL10] Norbert Hundeshagen and Peter Leupold. Transducing by Observing. In Henning Bordihn, Rudolf Freund, Markus Holzer, Thomas Hinze, Martin Kutrib, and Friedrich Otto, editors, *NCMA*, volume 263 of *books@ocg.at*, pages 85–98. Österreichische Computer Gesellschaft, 2010.

[HL12] Norbert Hundeshagen and Peter Leupold. Transducing by Observing and Restarting Transducers. In Rudolf Freund, Markus Holzer, Bianca Truthe, and Ulrich Ultes-Nitsche, editors, *NCMA*, volume 290 of *books@ocg.at*, pages 93–106. Österreichische Computer Gesellschaft, 2012.

[HL13] Norbert Hundeshagen and Peter Leupold. Transducing by Observing Length-Reducing and Painter Rules. Submitted for publication, January 2013.

[HO11] Norbert Hundeshagen and Friedrich Otto. Characterizing the Regular Languages by Nonforgetting Restarting Automata. In Giancarlo Mauri and Alberto Leporati, editors, *Developments in Language Theory*, volume 6795 of *Lecture Notes in Computer Science*, pages 288–299. Springer, 2011.

[HO12a] Norbert Hundeshagen and Friedrich Otto. Characterizing the Rational Functions by Restarting Transducers. In Adrian Horia Dediu and Carlos Martín-Vide, editors, *LATA*, volume 7183 of *Lecture Notes in Computer Science*, pages 325–336. Springer, 2012.

[HO12b] Norbert Hundeshagen and Friedrich Otto. Restarting Transducers, Regular Languages, and Rational Relations. Submitted for publication, June 2012.

[HOV10] Norbert Hundeshagen, Friedrich Otto, and Marcel Vollweiler. Transductions Computed by PC-Systems of Monotone Deterministic Restarting Automata. In Michael Domaratzki and Kai Salomaa, editors, *CIAA*, volume 6482 of *Lecture Notes in Computer Science*, pages 163–172. Springer, 2010.

[HU69] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1969.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[IC73] Karel Culik II and Rina S. Cohen. LR-Regular Grammars - an Extension of LR(k) Grammars. *J. Comput. Syst. Sci.*, 7(1):66–96, 1973.

[JL02] Tomasz Jurdzinski and Krzysztof Lorys. Church-Rosser Languages vs. UCFL. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, editors, *ICALP*, volume 2380 of *Lecture Notes in Computer Science*, pages 147–158. Springer, 2002.

[JM09] Dan Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, Englewood Cliffs, NJ, 2. edition, 2009.

[JMOP06] Tomasz Jurdzinski, Frantisek Mráz, Friedrich Otto, and Martin Plátek. Degrees of Non-Monotonicity for Restarting Automata. *Theor. Comput. Sci.*, 369(1-3):1–34, 2006.

[JMPV95] Petr Jancar, Frantisek Mráz, Martin Plátek, and Jörg Vogel. Restarting Automata. In Horst Reichel, editor, *FCT*, volume 965 of *Lecture Notes in Computer Science*, pages 283–292. Springer, 1995.

[JMPV97] Petr Jancar, Frantisek Mráz, Martin Plátek, and Jörg Vogel. On Restarting Automata with Rewriting. In Gheorghe Paun and Arto Salomaa, editors, *New Trends in Formal Languages*, volume 1218 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 1997.

[JMPV98] Petr Jancar, Frantisek Mráz, Martin Plátek, and Jörg Vogel. Different Types of Monotonicity for Restarting Automata. In Vikraman Arvind and Ramaswamy Ramanujam, editors, *FSTTCS*, volume 1530 of *Lecture Notes in Computer Science*, pages 343–354. Springer, 1998.

[JMPV99] Petr Jancar, Frantisek Mráz, Martin Plátek, and Jörg Vogel. On Monotonic Automata with a Restart Operation. *Journal of Automata, Languages and Combinatorics*, 4(4):287–312, 1999.

[JO07] Tomasz Jurdzinski and Friedrich Otto. Shrinking Restarting Automata. *Int. J. Found. Comput. Sci.*, 18(2):361–385, 2007.

[KK94] Ronald M. Kaplan and Martin Kay. Regular Models of Phonological Rule Systems. *Computational Linguistics*, 20(3):331–378, 1994.

[Kle56] Steven C. Kleene. Representation of Events in Nerve Nets and Finite Automata. In *Automata studies*, Annals of mathematics studies, no. 34, pages 3–41. Princeton University Press, Princeton, N. J., 1956.

[KO12] Martin Kutrib and Friedrich Otto. On the Descriptional Complexity of the Window Size for Deterministic Restarting Automata. In Nelma Moreira and Rogério Reis, editors, *CIAA*, volume 7381 of *Lecture Notes in Computer Science*, pages 253–264. Springer, 2012.

[KR08] Martin Kutrib and Jens Reimann. Succinct Description of Regular Languages by Weak Restarting Automata. *Inf. Comput.*, 206(9-10):1152–1160, 2008.

[LMP10] Markéta Lopatková, Frantisek Mráz, and Martin Plátek. Towards a Formal Model of Natural Language Description Based on Restarting Automata with Parallel DR-Structures. In Dana Pardubská, editor, *ITAT*, volume 683 of *CEUR Workshop Proceedings*, pages 25–32. CEUR-WS.org, 2010.

[LPK05] Markéta Lopatková, Martin Plátek, and Vladislav Kubon. Modeling Syntax of Free Word-Order Languages: Dependency Analysis by Reduction. In Václav Matousek, Pavel Mautner, and Tomás Pavelka, editors, *TSD*, Lecture Notes in Computer Science, pages 140–147. Springer, 2005.

[LPS07] Markéta Lopatková, Martin Plátek, and Petr Sgall. Towards a Formal Model for Functional Generative Description: Analysis by Reduction and Restarting Automata. *Prague Bull. Math. Linguistics*, 87:7–26, 2007.

[McK64] J. D. McKnight, Jr. Kleene Quotient Theorems. *Pacific J. Math.*, 14:1343–1352, 1964.

[Mea55] George H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[Mes08] Hartmut Messerschmidt. *CD-Systems of Restarting Automata*. PhD thesis, Fachbereich Elektrotechnik/Informatik, Universität Kassel, 2008.

[MNO88] Robert McNaughton, Paliath Narendran, and Friedrich Otto. Church-Rosser Thue Systems and Formal Languages. *J. ACM*, 35(2):324–344, 1988.

[MO11] Hartmut Messerschmidt and Friedrich Otto. A Hierarchy of Monotone Deterministic Non-Forgetting Restarting Automata. *Theory Comput. Syst.*, 48(2):343–373, 2011.

[Moh97] Mehryar Mohri. Finite-State Transducers in Language and Speech Processing. *Computational Linguistics*, 23(2):269–311, 1997.

[Moo56] Edward F. Moore. Gedanken Experiments on Sequential Machines. In *Automata Studies*, pages 129–153. Princeton U., 1956.

[Mrá01] Frantisek Mráz. Lookahead Hierarchies of Restarting Automata. *Journal of Automata, Languages and Combinatorics*, 6(4):493–506, 2001.

[MS04] Hartmut Messerschmidt and Heiko Stamer. Restart-Automaten mit mehreren Restart-Zuständen. In H. Bordihn, editor, *Workshop 'Formale Sprachen in der Linguistik' und 14. Theorietag 'Automaten und Formale Sprachen', Proc.*, pages 111–116, Institut für Informatik, Universität Potsdam, 2004.

[Nie02] Gundula Niemann. *Church-Rosser Languages and Related Classes.* PhD thesis, Fachbereich Mathematik/Informatik, Universität Kassel, 2002.

[Niv68] Maurice Nivat. Transductions des Langages de Chomsky. *Ann. Inst. Fourier*, 18(1):339–456, 1968.

[NO99] Gundula Niemann and Friedrich Otto. Restarting Automata, Church-Rosser Languages, and Representations of R.E. Languages. In Grzegorz Rozenberg and Wolfgang Thomas, editors, *Developments in Language Theory*, pages 103–114. World Scientific, 1999.

[NO00] Gundula Niemann and Friedrich Otto. Further Results on Restarting Automata. In Masami Ito and Teruo Imaoka, editors, *Words, Languages & Combinatorics*, pages 352–369, 2000.

[NO01] Gundula Niemann and Friedrich Otto. On the Power of RRWW-Automata. In Masami Ito, Gheorghe Paun, and Sheng Yu, editors, *Words, Semigroups, and Transductions*, pages 341–355. World Scientific, 2001.

[NO12] Benedek Nagy and Friedrich Otto. On CD-Systems of Stateless Deterministic R-Automata with Window Size One. *J. Comput. Syst. Sci.*, 78(3):780–806, 2012.

[Ott] Friedrich Otto. Formale Sprachen und Automaten. Skript zur gleichnamigen Vorlesung, Universität Kassel.

[Ott06] Friedrich Otto. Restarting Automata. In Zoltán Ésik, Carlos Martín-Vide, and Victor Mitrana, editors, *Recent Advances in Formal Languages and Applications*, volume 25, pages 269–303. Springer, 2006.

[Ott10] Friedrich Otto. On Proper Languages and Transformations of Lexicalized Types of Automata. In M. Itō, Y. Kobayashi, and K. Shoji, editors, *Automata, Formal Languages and Algebraic Systems - Proceedings of Aflas 2008*, pages 201–222. World Scientific, 2010.

[Plá01] Martin Plátek. Two-Way Restarting Automata and J-Monotonicity. In Leszek Pacholski and Peter Ruzicka, editors, *SOFSEM*, volume 2234 of *Lecture Notes in Computer Science*, pages 316–325. Springer, 2001.

[PLO03] Martin Plátek, Markéta Lopatková, and Karel Oliva. Restarting Automata: Motivations and Applications. In M. Holzer, editor, *13. Theorietag 'Automaten und Formale Sprachen', Proc.*, pages 90 – 96, Technische Universität München, 2003.

[PML10a] Martin Plátek, Frantisek Mráz, and Markéta Lopatková. (In)Dependencies in Functional Generative Description by Restarting Automata. In Henning Bordihn, Rudolf Freund, Markus Holzer, Thomas Hinze, Martin Kutrib, and Friedrich Otto, editors, *NCMA*, volume 263 of *books@ocg.at*, pages 155–170. Austrian Computer Society, 2010.

[PML10b] Martin Plátek, Frantisek Mráz, and Markéta Lopatková. Restarting Automata with Structured Output and Functional Generative Description. In Adrian Horia Dediu, Henning Fernau, and Carlos Martín-Vide, editors, *LATA*, volume 6031 of *Lecture Notes in Computer Science*, pages 500–511. Springer, 2010.

[Rei07] Jens Reimann. *Beschreibungskomplexität von Restart-Automaten.* PhD thesis, Naturwissenschaftliche Fachbereiche, Justus-Liebig-Universität Giessen, 2007.

[RS59] Michael O. Rabin and Dana Scott. Finite Automata and Their Decision Problems. *IBM J. Res. Dev.*, 3(2):114–125, April 1959.

[RS96] Emmanuel Roche and Yves Schabes. Introduction to Finite-State Devices in Natural Language Processing. Technical report, Mitsubishi Electric Research Laboratories, 1996.

[RS97] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages, Vol. 1: Word, Language, Grammar.* Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[RS08] Jean-François Raskin and Frédéric Servais. Visibly Pushdown Transducers. In *ICALP (2)*, pages 386–397, 2008.

[Sak09] Jacques Sakarovitch. *Elements of Automata Theory.* Cambridge University Press, 2009.

[San04] Nicolae Santean. Bimachines and Structurally-Reversed Automata. *Journal of Automata, Languages and Combinatorics*, 9(1):121–146, 2004.

[Sch75] Marcel Paul Schützenberger. Sur les Relations Rationnelles. In H. Brakhage, editor, *Automata Theory and Formal Languages*, volume 33 of *Lecture Notes in Computer Science*, pages 209–213. Springer, 1975.

[Sch10] Natalie Schluter. On Lookahead Hierarchies for Monotone and Deterministic Restarting Automata with Auxiliary Symbols (Extended Abstract). In Yuan Gao, Hanlin Lu, Shinnosuke Seki, and Sheng Yu, editors, *Developments in Language Theory*, volume 6224 of *Lecture Notes in Computer Science*, pages 440–441. Springer, 2010.

[Sén99] Géraud Sénizergues. T(A) = T(B)? In Jirí Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *ICALP*, volume 1644 of *Lecture Notes in Computer Science*, pages 665–675. Springer, 1999.

[Sta08] Heiko Stamer. *Restarting Tree Automata*. PhD thesis, Fachbereich Elektrotechnik/Informatik, Universität Kassel, 2008.

[VO12] Marcel Vollweiler and Friedrich Otto. Systems of Parallel Communicating Restarting Automata. In Rudolf Freund, Markus Holzer, Bianca Truthe, and Ulrich Ultes-Nitsche, editors, *NCMA*, volume 290 of *books@ocg.at*, pages 197–212. Österreichische Computer Gesellschaft, 2012.