Termination Analysis by Tree Automata Completion

Dissertation
in computer science

by
Martin Korp

submitted to the
Faculty of Mathematics, Computer Science and Physics
of the University of Innsbruck

in partial fulfillment of the requirements for the degree of
Doctor rerum naturalium

supervised by
Univ.-Prof. Dr. Aart Middeldorp
Institute of Computer Science

Innsbruck, June 2010

Abstract

Establishing *termination* of programs and processes is one of the most fundamental problems in computer science. In the area of *term rewriting*, a Turing-complete model of computation which forms the theoretical basis of functional programming, termination has been studied for several decades. Although termination is undecidable in general, many powerful termination criteria have been developed. In this dissertation we focus on methods that uses *automata techniques*, especially *tree automata completion*, to automatically prove the termination of rewrite systems.

A relatively new and elegant method within this scope is the so called *match-bound technique* proposed by Geser, Hofbauer, and Waldmann. It uses tree automata completion to prove the termination of rewrite systems. In this thesis we extend the match-bound technique in three directions. The first extension is the removal of the left-linearity restriction to increase the applicability of the method. The second extension is the integration of the match-bound technique into the dependency pair framework, a powerful framework to automatically prove the termination of rewrite systems. The third extension discusses how the match-bound technique can be used for *modular complexity analysis*.

Another termination technique that can benefit from the use of automata techniques is the so called *dependency graph processor*, which is one of the most frequently used techniques within the dependency pair framework. We illustrate that by using tree automata completion tremendous gains in power can be achieved.

The developed techniques have been integrated into the tool $\mathsf{T_TT_2}$, a fully automatic termination analyzer for first-order term rewrite systems, as well as the complexity analyzer $\mathsf{C_aT}$.

Acknowledgments

First of all I would like to express my gratitude to my adviser Aart Middeldorp who made it possible for me to write this thesis. His valuable and informative guidance was definitely one of the decisive factors for the success and progress of this thesis. During our cooperation I could learn a lot from him and I am thankful for those experiences.

Special gratitude is also dedicated to Christian Sternagel and Harald Zankl. I shared with them an office during the first three years of my Ph.D. studies and it was mainly their merit that I felt very comfortable during work. I enjoyed the discussions concerning research, TTT2, and CaT (although some of them where quite heavy) but also all common activities outside the office. So thanks again to both of you. I will always gladly remember the good old days in 3M09. Apart from Christian and Harald I would like to thank Friedrich Neurauter and Sarah Winkler who shared an office with me during the last year of my Ph.D. studies. I had a nice time in 3M03 and the pleasure to be one of the founders of the legendary CL Lab Parties.

Of course I would like to thank also all other members of the Computational Logic group, namely Martin Avanzini, Simon Bailey, Martina Ingenhaeff, Georg Moser, Andreas Schnabl, and René Thiemann, for useful and constructive discussion and for providing a great group atmosphere.

Last but not least, I would like to thank my family, especially my parents Rosa and Rudi, my brother Matthias, and my better half Maria, who have always supported me. Without you, all this would certainly not have been possible.

Contents

Progress is possible only if we train ourselves to think about programs without thinking of them as pieces of executable code.

Edsger Dijkstra

Chapter 1

Introduction

One of the milestones in the field of theoretical computer science was the foundation of *computability theory* by Church and Turing in the late 1930s. Inspired by Hilbert's Entscheidungsproblem, Turing and Church where the first who attempted to make the notion of computability precise, albeit from different points of view. While Church developed the so called λ-calculus [5] to characterize computable functions, Turing followed a more mechanical approach which resulted in the definition of Turing machines [57]. Shortly after the publication of the two achievements Turing showed that both approaches capture the same notion of computability [58]. Based on this result, the famous and well-known *Church-Turing thesis* arose. It basically states that any computable function can be expressed as a λ-term or, alternatively, in form of a Turing machine. Although this thesis cannot be proved formally because the notion of what it means for a function to be computable is vague and intuitive, all models of computation yet discovered have been shown to be equivalent to these early models of computation.

In this thesis we are concerned with another Turing-complete model of computation called *term rewriting*. Term rewriting is a branch of theoretical computer science which forms the theoretical fundament of functional programming and theorem proving. Its foundation is equational logic. What distinguishes term rewriting from equational logic is that equations are used as directed rules to perform computations. Although the origins of rewriting can be traced back to mathematics and mathematical logic, it took until the early 20$^{\text{th}}$ century for the development of term rewriting to start. In 1914, Thue used for the first time rewrite systems to manipulate strings [56]. Later on, in the 1930s, the introduction of the λ-calculus and its twin combinatory logic led to fundamental results in the area of term rewriting [2, 54]. Shortly after, the notion of a term rewrite system in its present form was formulated and since then used in various applications like type specifications, theorem proving, or programming languages.

An important and interesting research field within the area of term rewriting is concerned with the analysis of the *termination* behavior of term rewrite systems. Establishing termination of programs and processes is one of the most fundamental problems in computer science. In the area of term rewriting, termination has been studied for several decades. The simplicity as well as its relation to functional programming languages makes term rewriting a perfect environment for the study of termination. Although termination is undecidable in general [2, 54], many powerful termination criteria have been developed. The first techniques have been published in the early 1970s by Manna and Ness as well as Knuth and Bendix. In contrast to the theoretical work of Manna and Ness, in which termination has been characterized by using *reduction orderings* [47], Knuth and Bendix developed the first automatable termination criterion—the well-known *Knuth-Bendix order* [38]. In the aftermath of those trend-setting papers, *simplification orderings* dominated research for many years. It took until the early 1990s when the development of termination techniques which overcome the limitations of simplification orderings became the main focus of attention. Some prominent techniques which emerged at the beginning of this phase are *transformation orders*, introduced by Bellegarde and Lescanne [3], and *semantic labeling*, developed by Zantema [62]. Ultimately in 2000, Arts and Giesl introduced the so called *dependency pair method* [1] which immediately shaped up as one of the most powerful techniques, till this day.

At the beginning of the 21$^{\text{st}}$ century, a significant change of the nature of termination analysis took place. Instead of developing new techniques, the focus turned on automation of existing termination criteria. In 2004, the first *international termination competition* took place.[1] Since then, this event has been annually repeated. In the last edition of the termination competition in 2009, a dozen of termination tools competed against each other in various categories.

1.1 Motivation and Outline

One reason for the importance of termination analysis is that it can be used to check the correctness of programs and processes. It is often the case that properties which guarantee a correct behavior of safety-critical systems can be automatically transformed into suitable termination problems. A quite common and alternative approach to verify the correctness of a program is model-checking. In the area of term rewriting, model-checking is usually performed by analyzing the terms that can be reached via rewriting from some initial terms. In this thesis we combine both techniques to a certain extend to prove the correctness of term rewrite systems by analyzing their termination behavior. To perform reachability analysis we use *tree automata completion* which represents an elegant and efficient method to compute all terms that can be reached from some initial terms via rewriting. Thereby tree automata are used to represent possibly infinite sets of terms. As initial point we consider the *match-bound technique* proposed by Geser, Hofbauer, and Waldmann in [21]. It is a relatively new and elegant termination method which combines elements of semantic labeling and simplification orders to prove the termination of term rewrite systems. In order to obtain a termination certificate it uses automata techniques, in particular, tree automata completion.

In this thesis we extend the match-bound technique in three directions. At first we remove the *left-linearity* restriction to increase the applicability of the method. To this end we introduce a new approach to cope with non-left-

[1]`http://termination-portal.org/`

linear rewrite rules during tree automata completion. Secondly we show how the match-bound technique can be integrated into the *dependency pair framework* [27], a modular reformulation and improvement of the dependency pair method [1]. To guarantee a successful cooperation with other techniques within this framework, we restructure the match-bound technique in such a way that the termination of a single rewrite rule, relative to all other rules, can be proved. Last but not least, we discuss how the match-bound technique can be used for *modular complexity analysis*. To achieve this goal we switch from termination to relative termination and present an enhanced version of relative match-bounds.

Another termination technique that can benefit from the use of tree automata techniques is the so called *dependency graph processor* [28, 48]. This method is one of the most frequently used techniques within the dependency pair framework. We show that by using tree automata completion tremendous gains in power can be achieved.

Most of the results presented in this thesis appeared already in the conference proceedings [40, 41, 42, 44, 61] as well as in the journal article [43]. In this thesis we slightly improve and extend these contributions. Besides a more detailed presentation of the results we additionally explain how the developed techniques can and have been implemented in the termination prover TTT2 as well as the complexity tool CaT.

1.2 Structure

The remainder of the thesis is organized as follows. In the next chapter we recall some basic definitions concerning term rewriting and tree automata. Chapter 3 is devoted to tree automata completion. Besides the classical approach we present a new and elegant way to deal with non-left-linear rewrite rules, using so called *quasi-deterministic* tree automata. Afterwards, in Chapter 4, we introduce the *match-bound technique* and its extension to non-left-linear rewrite rules. In Chapter 5 we first recall some basic definitions concerning the dependency pair framework. After that we introduce the concept of *e(-raise)-DP-bounds* which ensures a fully modular integration of the match-bound technique into the dependency pair framework. Finally we discuss how the *dependency graph processor* can be improved using tree automata completion. Chapter 6 is concerned with *complexity analysis*. Besides the presentation of a modular framework, which allows us to infer bounds on the complexity of term rewrite systems by combining different criteria, we show how the match-bound technique can be adapted such that it fits into this framework. All techniques presented in the preceding chapters have been integrated in the termination prover TTT2 as well as the complexity tool CaT. In Chapter 7 we report on the extensive experiments that we conducted. Finally in Chapter 8 we conclude. Some additional material concerning TTT2 and CaT as well as some of the more technical proofs are deferred to Appendices A and B.

Chapter 2

Preliminaries

We assume familiarity with term rewriting [2] and tree automata [6]. Below we recall some important definitions needed in the remainder of the thesis.

2.1 Term Rewriting

A *signature* consists of function symbols equipped with fixed arities. The set of terms constructed from a signature $\mathcal{F}$ and a set of variables $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F},\mathcal{V})$. Likewise, we write $\mathcal{T}(\mathcal{F})$ for the set of ground terms induced by the signature $\mathcal{F}$. The set of variables occurring in a term t is denoted by $\mathcal{V}\mathsf{ar}(t)$ and the set of function symbols of t is denoted by $\mathcal{F}\mathsf{un}(t)$. The size of a term t is denoted by $|t|$ and inductively defined as $|t| = 1$ if t is a variable and $|t| = 1+|t_1|+\cdots+|t_n|$ if $t = f(t_1,\ldots,t_n)$. The number of function symbols occurring in a term t is abbreviated by $\|t\|$. We write $\mathsf{depth}(t)$ to refer to the *depth* of a term t which is inductively defined as $\mathsf{depth}(t) = 0$ if t is a variable or a constant and $\mathsf{depth}(t) = 1+\max\{\mathsf{depth}(t_i) \mid i \in \{1,\ldots,n\}\}$ if $t = f(t_1,\ldots,t_n)$. A term t is called *linear* if each variable in $\mathcal{V}\mathsf{ar}(t)$ occurs exactly once in t. Positions are used to address symbol occurrences in terms. The set of positions induced by a term t is denoted by $\mathcal{P}\mathsf{os}(t)$ and inductively defined as follows: $\mathcal{P}\mathsf{os}(t) = \{\epsilon\}$ if t is a variable and $\mathcal{P}\mathsf{os}(t) = \{\epsilon\} \cup \{ip \mid i \in \{1,\ldots,n\} \text{ and } p \in \mathcal{P}\mathsf{os}(t_i)\}$ if $t = f(t_1,\ldots,t_n)$. Given a term t and a position $p \in \mathcal{P}\mathsf{os}(t)$, we write $t(p)$ for the function symbol or variable at position p. In case that $p = \epsilon$ we often write $\mathsf{root}(t)$ to refer to the root symbol of the term t. We use $\mathcal{P}\mathsf{os}_{\mathcal{F}}(t)$ to denote the subset of positions $p \in \mathcal{P}\mathsf{os}(t)$ such that $t(p)$ is a function symbol. Let s and t be two terms. We say that s is a *subterm* of t, written $s \trianglelefteq t$, if either $s = t$ or $t = f(t_1,\ldots,t_n)$ and s is a subterm of t_i for some $i \in \{1,\ldots,n\}$. A subterm s of t is *proper*, denoted by $s \lhd t$, if $s \neq t$. We write $t|_p$ to denote the subterm of t at position p and $s[t]_p$ to denote the term obtained from s by replacing the subterm at position p by the term t.

Contexts are terms over the extended signature $\mathcal{F} \cup \{\Box\}$ with exactly one occurrence of the fresh constant $\Box$ (also called hole). The expression $C[t]$ denotes the term obtained from the context C and the term t by replacing the hole $\Box$ in C by t. We extend contexts to *multi-contexts* (contexts with more than one hole) in the usual manner. A *substitution* σ is a mapping from variables to terms. As usual we assume that the domain of σ, denoted by $\mathcal{D}\mathsf{om}(\sigma)$, is finite. The application of a substitution σ to a term t, written as $t\sigma$, is defined as $t\sigma = \sigma(t)$ if t is a variable and $t\sigma = f(t_1\sigma,\ldots,t_n\sigma)$ if $t = f(t_1,\ldots,t_n)$.

A *rewrite rule* is a pair of terms (l, r), written $l \to r$. A rewrite rule $l \to r$ is called *duplicating* if there is a variable $x \in \mathcal{V}\mathsf{ar}(r)$ which occurs more often in r than in l. We say that a rewrite rule $l \to r$ is *collapsing* if r is a variable. A rewrite rule $l \to r$ is called *linear* if both l and r are linear. Similarly, $l \to r$ is called *left-linear* (*right-linear*) if l (r) is linear. A *term rewrite system* (TRS for short) $\mathcal{R}$ is a set of rewrite rules such that for all $l \to r \in \mathcal{R}$, l is not a variable and $\mathcal{V}\mathsf{ar}(l) \supseteq \mathcal{V}\mathsf{ar}(r)$. The *defined symbols* of a TRS $\mathcal{R}$ are all function symbols f for which there is a rewrite rule $l \to r$ in $\mathcal{R}$ such that $f = \mathsf{root}(l)$. In the following we denote this set of function symbols by $\mathcal{F}\mathsf{un}_{\mathcal{D}}(\mathcal{R})$. Those function symbols of $\mathcal{R}$ which are not defined are called *constructor symbols*. So the set of all constructor symbols is defined as $\mathcal{F}\mathsf{un}_{\mathcal{C}}(\mathcal{R}) = \mathcal{F} \setminus \mathcal{F}\mathsf{un}_{\mathcal{D}}(\mathcal{R})$. Here $\mathcal{F}$ denotes the signature of $\mathcal{R}$. A TRS $\mathcal{R}$ is called *duplicating* (*collapsing*) if at least one rewrite rule in $\mathcal{R}$ is duplicating (collapsing). Likewise, a TRS $\mathcal{R}$ is called *linear* (*left-linear*, *right-linear*) if all rewrite rules in $\mathcal{R}$ are linear (left-linear, right-linear respectively). Given some TRS $\mathcal{R}$, we write $\mathcal{R}^{-1}$ to denote the set $\{r \to l \mid l \to r \in \mathcal{R}\}$ of rewrite rules. Note that $\mathcal{R}^{-1}$ is not necessarily a TRS. A *rewrite relation* is a binary relation on terms that is closed under contexts and substitutions. For a set of rewrite rules $\mathcal{R}$ we define $\to_{\mathcal{R}}$ to be the smallest rewrite relation that contains $\mathcal{R}$, that is, $s \to_{\mathcal{R}} t$ for two terms s and t if and only if there is a rewrite rule $l \to r \in \mathcal{R}$, a position $p \in \mathcal{P}\mathsf{os}(s)$, and a substitution σ such that $s = s[l\sigma]_p$ and $t = s[r\sigma]_p$. As usual $\to_{\mathcal{R}}^{+}$ denotes the transitive and $\to_{\mathcal{R}}^{*}$ the reflexive and transitive closure of $\to_{\mathcal{R}}$. In the following we drop the subscript $\mathcal{R}$ from $\to_{\mathcal{R}}$ and its derivatives when no confusion about $\mathcal{R}$ can arise. We say that a term s is a *normal form* with respect to a TRS $\mathcal{R}$ if there is no term t such that $s \to_{\mathcal{R}} t$. The set of all normal forms of a TRS $\mathcal{R}$ is denoted by $\mathsf{NF}(\mathcal{R})$. Let $\mathcal{R}$ be a set of rewrite rules over a signature $\mathcal{F}$. A *rewrite sequence* or *derivation* is a possibly infinite sequence of $\to_{\mathcal{R}}$-steps. We say that $\mathcal{R}$ is *terminating* if $\to_{\mathcal{R}}$ is well-founded, that is, if $\mathcal{R}$ does not admit an infinite rewrite sequence. Given a set $L \subseteq \mathcal{T}(\mathcal{F})$ of ground terms, we say that $\mathcal{R}$ is *terminating on* L if none of the terms in L admit an infinite rewrite sequence. Last but not least, the set $\{t \in \mathcal{T}(\mathcal{F}) \mid s \to_{\mathcal{R}}^{*} t \text{ for some } s \in L\}$ of descendants of L is denoted by $\to_{\mathcal{R}}^{*}(L)$.

Example 2.1. Consider the TRS $\mathcal{R}$ consisting of the rewrite rules

$$\begin{aligned} \mathsf{0} + y &\to y & \mathsf{0} \times y &\to \mathsf{0} \\ \mathsf{s}(x) + y &\to \mathsf{s}(x + y) & \mathsf{s}(x) \times y &\to (x \times y) + y \end{aligned}$$

which specify addition and multiplication over natural numbers in unary notation. By applying the above rules we can easily compute the result of the term $t = (\mathsf{s}(\mathsf{0}) + \mathsf{0}) \times \mathsf{s}(\mathsf{s}(\mathsf{0}))$:

$$\begin{aligned} t &\to_{\mathcal{R}} \mathsf{s}(\mathsf{0} + \mathsf{0}) \times \mathsf{s}(\mathsf{s}(\mathsf{0})) \to_{\mathcal{R}} \mathsf{s}(\mathsf{0}) \times \mathsf{s}(\mathsf{s}(\mathsf{0})) \\ &\to_{\mathcal{R}} (\mathsf{0} \times \mathsf{s}(\mathsf{s}(\mathsf{0}))) + \mathsf{s}(\mathsf{s}(\mathsf{0})) \to_{\mathcal{R}} \mathsf{0} + \mathsf{s}(\mathsf{s}(\mathsf{0})) \\ &\to_{\mathcal{R}} \mathsf{s}(\mathsf{s}(\mathsf{0})) \end{aligned}$$

For instance, in the first rewrite step the rewrite rule $\mathsf{s}(x) + y \to \mathsf{s}(x + y)$ is applied to the term t at position 1, to reduce t to $\mathsf{s}(\mathsf{0} + \mathsf{0}) \times \mathsf{s}(\mathsf{s}(\mathsf{0}))$. The corresponding matching substitution is $\sigma = \{x \mapsto \mathsf{0}, y \mapsto \mathsf{0}\}$. Since the resulting term $\mathsf{s}(\mathsf{s}(\mathsf{0}))$ does not permit any further rewrite steps, it is a normal form of the TRS $\mathcal{R}$. It is easy to see that the ground normal forms of $\mathcal{R}$ are all terms which do not contain $+$ and $\times$. So $\mathsf{NF}(\mathcal{R}) \supseteq \mathcal{T}(\{\mathsf{0}, \mathsf{s}\})$. Additionally, $\mathsf{NF}(\mathcal{R})$ also contains terms like $x + y$, $(x + y) + z$, etc.

In the following we assume that every signature $\mathcal{F}$ contains at least one constant, $L \subseteq \mathcal{T}(\mathcal{F})$ denotes a possibly infinite set of ground terms (also called language) induced by the signature $\mathcal{F}$ of the underlying system, and all TRSs are finite unless it is indicated otherwise.

2.2 Tree Automata

A *tree automaton* $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ consists of a finite signature $\mathcal{F}$, a finite set of states Q, a set of final states $Q_f \subseteq Q$, and a set of transitions Δ which are either of the form $f(q_1, \ldots, q_n) \to q$ or $p \to q$ where f is an n-ary function symbol in $\mathcal{F}$ and $p, q, q_1, \ldots, q_n \in Q$. Transitions of the latter form are called *ϵ-transitions*. In the following we often write $l \to q_1 \mid \cdots \mid q_n$ to abbreviate that Δ contains transitions $l \to q_1$, $\ldots$, $l \to q_n$. The rewrite relation induced by Δ is defined as $s \to_\Delta t$ if and only if there is a transition $l \to q \in \Delta$ and a position p such that $s|_p = l$ and $t = s[q]_p$. Similar as for TRSs we write $\to_\Delta^+$ and $\to_\Delta^*$ to denote the transitive as well as reflexive and transitive closure of $\to_\Delta$. Furthermore, we sometimes drop the subscript Δ from $\to_\Delta$ and its derivatives when no confusion about Δ can arise. We say that a term t is *accepted* by $\mathcal{A}$ if $t \to_\Delta^* q$ for some final state $q \in Q_f$. The *language* $\mathcal{L}(\mathcal{A})$ induced by $\mathcal{A}$ is defined as the set of all ground terms $t \in \mathcal{T}(\mathcal{F})$ which are accepted by $\mathcal{A}$. A tree automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ is called *deterministic* if it contains neither ϵ-transitions nor different transitions with the same left-hand side. Finally, a set of ground terms L is called *regular* if there exists a tree automaton $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = L$.

Example 2.2. Consider the tree automaton $\mathcal{A}$ with the transitions

$$\begin{array}{cccc} \mathsf{0} \to 1 & \mathsf{s}(1) \to 1 & \mathsf{s}(2) \to 2 & \times(2,1) \to 3 \\ +(1,1) \to 2 & +(2,1) \to 2 & +(1,2) \to 2 & +(2,2) \to 2 \end{array}$$

and the final state 3. The ground term $t = (\mathsf{s}(\mathsf{0}) + \mathsf{0}) \times \mathsf{0}$ induces the derivation

$$\begin{aligned} t &\to_\Delta (\mathsf{s}(1) + \mathsf{0}) \times \mathsf{0} \to_\Delta (1 + \mathsf{0}) \times \mathsf{0} \to_\Delta (1 + 1) \times \mathsf{0} \\ &\to_\Delta 2 \times \mathsf{0} \to_\Delta 2 \times 1 \to_\Delta 3 \end{aligned}$$

where, for instance in the fourth step the transition $+(1,1) \to 1$ is applied to the term $(1 + 1) \times \mathsf{0}$ at position 1. Since $t \to_\Delta^* 3$ and 3 is a final state we know that t is accepted by $\mathcal{A}$. It is not difficult to check that $\mathcal{A}$ accepts all ground terms of the form $u \times v$ such that $u \in \mathcal{T}(\{\mathsf{0}, \mathsf{s}, +\})$ contains at least one $+$ and $v \in \mathcal{T}(\{\mathsf{0}, \mathsf{s}\})$.

Chapter 3

Tree Automata Completion

Tree automata completion represents an elegant and efficient technique to perform reachability analysis. Starting from a regular language it aims to compute a regular superset of all terms that can be reached from a set of initial terms by performing rewriting. To represent possibly infinite sets of terms, *tree automata* are used. Since in general the set of descendants of a regular language and a rewrite system need not be regular, the challenge in this connection is to control the completion procedure in such a way that it terminates and the language accepted by the returned tree automaton is as exact as possible.

Initially, tree automata completion has been introduced for left-linear TRSs by Genet [18] and later on extended to non-left-linear TRSs by Genet and Tong [20]. Improvements and variations of the technique are discussed by Feuillade, Genet, and Viet Triem Tong in [14]. To obtain suitable approximations during the completion process, various *approximation techniques* have been developed. The most important ones are illustrated in [14, 20, 24]. The fact that tree automata completion has been used for example to certify properties of various cryptographic protocols [19, 52] or to prototype static analyzers in Java byte code [4], is a clear witness for the success of the approach. Nevertheless, a serious drawback of the technique still affects the treatment of *non-left-linear* rewrite rules. To ensure that rules of this kind can be handled, the overall idea is to constrain the underlying approximation technique such that critical transitions are kept deterministic. As a side effect the obtained over-approximations are not as exact as if arbitrary approximation techniques could be used.

The remainder of this chapter is organized as follows. After introducing tree automata completion for left-linear TRSs in Section 3.1, we discuss some variations and technical details of the approach in Sections 3.2 and 3.3. In Section 3.4 we present an alternative approach to cope with non-left-linear TRSs without limiting the underlying approximation techniques. To this end we use so called *quasi-deterministic* tree automata instead of non-deterministic tree automata during the completion process. Finally in Section 3.5 we show how quasi-deterministic tree automata can be efficiently constructed, especially, within the scope of the completion process.

Most of the results presented in this chapter appeared already in the conference paper [40] and the journal paper [43]. New contributions include an optimized definition of quasi-deterministic tree automata in Section 3.4 as well as the presentation of an advanced algorithm to construct quasi-deterministic tree automata, explained in Section 3.5.

3.1 Compatible Tree Automata

To construct a tree automaton $\mathcal{A}$ that accepts $\to_{\mathcal{R}}^*(L)$ for some left-linear TRS $\mathcal{R}$ and some language L we use *compatible* tree automata introduced in [24].

Definition 3.1. Let $\mathcal{R}$ be a left-linear TRS, $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ a tree automaton, and L a language. We say that $\mathcal{A}$ is *compatible* with $\mathcal{R}$ and L if $L \subseteq \mathcal{L}(\mathcal{A})$ and for each rewrite rule $l \to r \in \mathcal{R}$ and state substitution $\sigma\colon \mathcal{V}\mathsf{ar}(l) \to Q$ such that $l\sigma \to_{\Delta}^* q$ it holds that $r\sigma \to_{\Delta}^* q$.

Example 3.2. Consider the TRS $\mathcal{R}$ consisting of the rewrite rules

$$\mathsf{f}(x, y) \to \mathsf{f}(\mathsf{g}(x), \mathsf{g}(y)) \qquad\qquad \mathsf{h}(\mathsf{a}, y) \to \mathsf{h}(\mathsf{g}(y), \mathsf{g}(y))$$

and the tree automaton $\mathcal{A}$ with the final state 5 and the transitions

$$\begin{array}{lll} \mathsf{a} \to 1 & \mathsf{g}(1) \to 4 & \mathsf{f}(4,1) \to 5 \\ \mathsf{b} \to 2 & \mathsf{g}(4) \to 4 & \mathsf{h}(1,2) \to 5 \\ \mathsf{c} \to 3 & & \mathsf{h}(1,3) \to 5 \end{array}$$

accepting the terms $\mathsf{h}(\mathsf{a},\mathsf{b})$ and $\mathsf{h}(\mathsf{a},\mathsf{c})$ as well as all terms of the form $\mathsf{f}(\mathsf{g}^+(\mathsf{a}),\mathsf{a})$. Since $\mathsf{f}(x,y) \to_{\mathcal{R}} \mathsf{f}(\mathsf{g}(x),\mathsf{g}(y))$ and $\mathsf{f}(4,1) \to 5$ but $\mathsf{f}(\mathsf{g}(4),\mathsf{g}(1)) \not\to^* 5$, $\mathcal{A}$ is not compatible with $\mathcal{R}$ and $\mathcal{L}(\mathcal{A})$.

As the above definition already indicates, any compatible tree automaton $\mathcal{A}$ is closed under left-linear rewriting. The following result originates from [18].

Theorem 3.3. *Let $\mathcal{R}$ be a left-linear TRS and L a language. Let $\mathcal{A}$ be a tree automaton. If $\mathcal{A}$ is compatible with $\mathcal{R}$ and L then $\to_{\mathcal{R}}^*(L) \subseteq \mathcal{L}(\mathcal{A})$.* $\square$

So, as soon as we have constructed a tree automaton $\mathcal{A}$ that is compatible with some left-linear TRS $\mathcal{R}$ and a language L we can conclude that $\to_{\mathcal{R}}^*(L) \subseteq \mathcal{L}(\mathcal{A})$ by Theorem 3.3. Since the set $\to_{\mathcal{R}}^*(L)$ need not be regular, even for a linear TRS $\mathcal{R}$ and a regular language L [31], we cannot hope to give an exact automata construction.[1] So to obtain a tree automaton that is compatible with a TRS $\mathcal{R}$ and some language L, the general idea [18] is to start with some initial tree automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ accepting L and to look for violations of the compatibility requirement (see Figure 3.1): $l\sigma \to_{\Delta}^* q$ and $r\sigma \not\to_{\Delta}^* q$ for some rewrite rule $l \to r \in \mathcal{R}$, state substitution $\sigma\colon \mathcal{V}\mathsf{ar}(l) \to Q$, and state $q \in Q$. Then we add new states and transitions to the current automaton to ensure $r\sigma \to_{\Delta}^* q$. There are several ways to do this, ranging from establishing a completely new path $r\sigma \to_{\Delta}^* q$ to adding as few new transitions as possible by reusing transitions from the current automaton. After $r\sigma \to_{\Delta}^* q$ has been established, we look for further violations of compatibility. This process is repeated until a compatible automaton is obtained which may never happen if new states are kept being added.

It is obvious that we can always compute a compatible tree automaton which over-approximates the set $\to_{\mathcal{R}}^*(L)$ whenever $\mathcal{R}$ is a finite TRS over a finite

[1] For instance, for the TRS $\mathcal{R} = \{\mathsf{f}(x,y) \to \mathsf{f}(\mathsf{g}(x),\mathsf{g}(y))\}$ and the language $L = \{\mathsf{f}(\mathsf{a},\mathsf{a})\}$ we have $\to_{\mathcal{R}}^*(L) = \{\mathsf{f}(\mathsf{g}^n(\mathsf{a}),\mathsf{g}^n(\mathsf{a})) \mid n \geqslant 0\}$, which is clearly not regular.

Figure 3.1: Completion process

$$\begin{tikzcd} l\sigma \arrow[r, "*", "\mathcal{R}"'] \arrow[d, "\rhd"', "*"] & r\sigma \arrow[dl, bend left, "*"', "\rhd"] & \Longrightarrow & l\sigma \arrow[r, "*", "\mathcal{R}"'] \arrow[d, "\rhd"', "*"] & r\sigma \arrow[dl, bend left, dotted, "*"', "\rhd"] \\ q & & & q & \end{tikzcd}$$

signature. The challenge is to choose an approximation function that resolves compatibility violations in such a way that the completion process terminates and the language accepted by the returned tree automaton is as exact as possible. In Section 3.3 we discuss the most common approaches in that direction.

3.2 Detecting Compatibility Violations

Before we can solve compatibility violations and hence construct a compatible tree automaton we have to locate them. Because for a tree automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ and a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ we have in the worst case $|\mathcal{V}\mathsf{ar}(t)|^{|Q|}$ state substitutions σ such that $t\sigma \rightarrow_\Delta^* q$ for some $q \in Q$ it is obvious that each algorithm admits an exponential time behavior. In the literature, two approaches are proposed to detect compatibility violations. The first one, introduced in [17], uses a matching algorithm to enumerate all violations of the compatibility requirement whereas the second one, illustrated in [14], uses tree automata techniques to detect compatibility violations. Besides the fact that the two approaches use diverse techniques two locate compatibility violations, the main difference between the two approaches is that complexity wise, the second one is faster than the first one but also more difficult to implement. In the following we shortly introduce both approaches, beginning with the first one. To simplify the presentation we consider tree automata without ϵ-transitions.

3.2.1 Matching Algorithm

Let $\mathcal{F}$ be a signature and Q a set of states. A *matching problem* is a propositional formula made up of the propositions $\perp$ and $t \vdash s$ with $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and $s \in \mathcal{T}(\mathcal{F} \cup Q)$ as well as the connectives $\wedge$ and $\vee$. Here the states in Q are treated as constants. A state substitution σ is called a *solution* for a matching problem ϕ if $t\sigma \rightarrow_\Delta^* s$ when $\phi = t \vdash s$, σ is a solution for ϕ_1 and ϕ_2 if $\phi = \phi_1 \wedge \phi_2$, and σ is a solution for ϕ_1 or ϕ_2 if $\phi = \phi_1 \vee \phi_2$. The matching problem $\phi = \perp$ has no solution. To modify matching problems we use the transformation rules

$$\frac{t \vdash q}{t \vdash s_1 \vee \cdots \vee t \vdash s_n \vee \perp} \qquad \frac{f(t_1, \ldots, t_n) \vdash f(q_1, \ldots, q_n)}{t_1 \vdash q_1 \wedge \cdots \wedge t_n \vdash q_n}$$

where $t \notin \mathcal{V}$, $\mathsf{root}(t) = \mathsf{root}(s_i)$, and $s_i \rightarrow q \in \Delta$ for all $i \in \{1, \ldots, n\}$. Additionally we require that s_1, …, s_n represent all terms in $\mathsf{lhs}(\Delta)$ that satisfy the previous conditions. Besides the two transformation rules we use the simplification rules

$$\frac{\phi_1 \wedge (\phi_2 \vee \phi_3)}{(\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3)} \qquad\qquad \frac{\phi \wedge \bot}{\bot} \qquad\qquad \frac{\phi \vee \bot}{\phi}$$

to convert a matching problem into disjunctive normal form. Let $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ be a term and $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ a tree automaton. To compute all state substitutions σ such that $t\sigma \to_\Delta^* q$ for some $q \in Q$ we construct for each state q a matching problem $\phi = t \vdash q$ and transform it into a matching problem ϕ' by applying the above rules as long as possible. It is not difficult to see that either $\phi' = \bot$ or $\phi' = \phi_1 \vee \cdots \vee \phi_n$ and $\phi_i = x_{i_1} \vdash q_{i_1} \wedge \cdots \wedge x_{i_{n_i}} \vdash q_{i_{n_i}}$ with $x_{i_j} \in \mathcal{V}\mathsf{ar}(t)$ and $q_{i_j} \in Q$ for all $i \in \{1,\ldots,n\}$ and $j \in \{1,\ldots,n_i\}$. If $\phi' = \bot$ we conclude that ϕ has no solution and hence that there is no state substitution σ such that $t \to_\Delta^* q$. Otherwise, let $\sigma_i = \{x_{i_j} \mapsto q_{i_j} \mid j \in \{1,\ldots,n_i\}\}$. If σ_i is a valid state substitution, that is, $q_{i_j} = q_{i_k}$ whenever $x_{i_j} = x_{i_k}$ for all $j,k \in \{1,\ldots,n_i\}$, then σ_i is the unique solution of the matching problem ϕ_i. By the transformation rules applied to the matching problem ϕ it is guaranteed that ϕ_i is also a solution of the matching problem ϕ. Furthermore if $t\sigma \to_\Delta^* q$ for some substitution σ then $\sigma = \sigma_i$ for some $i \in \{1,\ldots,n\}$.

Example 3.4. Let $\mathcal{R}$ be the TRS and $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ the tree automaton of Example 3.2. Let $t = \mathsf{h}(\mathsf{a}, y)$ be the left-hand side of the second rewrite rule. We compute all state substitutions σ such that $t\sigma \to_\Delta^* 5$ using the above matching algorithm. We start with the initial matching problem $\mathsf{h}(\mathsf{a}, y) \vdash 5$. By applying the first transformation rule we obtain the new matching problem $\mathsf{h}(\mathsf{a}, y) \vdash \mathsf{h}(1,2) \vee \mathsf{h}(\mathsf{a}, y) \vdash \mathsf{h}(1,3) \vee \bot$. Next we apply two times the second transformation rule yielding the new problem $(\mathsf{a} \vdash 1 \wedge y \vdash 2) \vee (\mathsf{a} \vdash 1 \wedge y \vdash 3) \vee \bot$. After that we can apply two times the first transformation rule producing the matching problem $((\mathsf{a} \vdash \mathsf{a} \vee \bot) \wedge y \vdash 2) \vee ((\mathsf{a} \vdash \mathsf{a} \vee \bot) \wedge y \vdash 3) \vee \bot$. Finally, with the second transformation rule and the third simplification rule we can reduce the problem to $y \vdash 2 \vee y \vdash 3$. The solutions for this matching problem are $\sigma_1 = \{y \mapsto 2\}$ and $\sigma_2 = \{y \mapsto 3\}$. It follows that $t\sigma \to_\Delta^* 5$ if and only if $\sigma = \sigma_1$ or $\sigma = \sigma_2$.

3.2.2 Using Tree Automata Techniques

Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a tree automaton, $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ a term, and $q \in Q$ a state. In order to compute all possible state substitutions σ such that $t\sigma \to_\Delta^* q$ for some $q \in Q$ we first construct a tree automaton $\mathcal{A}_t$ over the extended signature $\mathcal{F} \cup \mathcal{V}\mathsf{ar}(t)$ which accepts the term t. Here variables contained in $\mathcal{V}\mathsf{ar}(t)$ are treated as constants. We define $\mathcal{A}_t = (\mathcal{F} \cup \mathcal{V}\mathsf{ar}(t), Q', Q'_f, \Delta')$ where $Q' = \{q_s \mid s \trianglelefteq t\}$ and $Q'_f = \{q_t\}$. Furthermore, for each subterm $s \trianglelefteq t$ with $s = f(s_1,\ldots,s_n)$, we require that the transition $f(q_{s_1},\ldots,q_{s_n}) \to q_s$ belongs to Δ'. It is easy to see that $\mathcal{L}(\mathcal{A}_t) = \{t\}$. Using the tree automaton $\mathcal{A}_t$ we can now construct a tree automaton $\mathcal{A}_\sigma = (\mathcal{F} \cup \mathcal{V}\mathsf{ar}(t), Q'', Q''_f, \Delta'')$ which accepts all state substitutions σ such that $t\sigma \to_\Delta^* q$ for some $q \in Q$. To this end we define $Q'' = Q' \times Q$ and $Q''_f = Q'_f \times Q$. Furthermore we have

$x \to (q_x, q) \in \Delta''$ for all transitions $x \to q_x \in \Delta'$ and states $q \in Q$ as well as $f((p_1, q_1), \ldots, (p_n, q_n)) \to (p, q) \in \Delta''$ for all transitions $f(p_1, \ldots, p_n) \to p \in \Delta'$ and $f(q_1, \ldots, q_n) \to q \in \Delta$. According to the construction of $\mathcal{A}_\sigma$ there is a state substitution σ such that $t\sigma \to^*_\Delta q$ for some $q \in Q$ if and only if we have $t \to^*_{\Delta''} (q_t, q)$ where each variable $x \in \mathcal{V}\mathsf{ar}(t)$ is replaced by the state $(q_x, x\sigma)$.

Example 3.5. Consider the TRS $\mathcal{R}$ and the tree automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ over the signature $\mathcal{F} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{f}, \mathsf{g}, \mathsf{h}\}$ of Example 3.2. Let $t = \mathsf{h}(\mathsf{a}, y)$ be the left-hand side of the second rewrite rule. To compute all potential compatibility violations caused by the rewrite rule $\mathsf{h}(\mathsf{a}, y) \to \mathsf{h}(\mathsf{g}(y), \mathsf{g}(y))$ using the above approach we first construct the tree automaton $\mathcal{A}_t = (\mathcal{F} \cup \mathcal{V}\mathsf{ar}(t), Q', Q'_f, \Delta')$ which accepts the term t. The above procedure yields $Q' = \{1, 2, 3\}$, $Q'_f = \{3\}$, and Δ' consisting of the transitions

$$\mathsf{a} \to 1 \qquad\qquad y \to 2 \qquad\qquad \mathsf{h}(1, 2) \to 3$$

where $q_\mathsf{a} = 1$, $q_y = 2$, and $q_{\mathsf{h}(\mathsf{a},y)} = 3$. Next we compute the tree automaton $\mathcal{A}_\sigma = (\mathcal{F} \cup \mathcal{V}\mathsf{ar}(t), Q'', Q''_f, \Delta'')$. To simplify the presentation we restrict Q'' to those states that occur at some right-hand side. (Note that all other states can be ignored because they can never occur in any $\to_\Delta$ sequence.) Hence we have $Q'' = \{(1, 1), (2, 2), (2, 3), (3, 5)\}$, $Q''_f = \{(3, 5)\}$, and Δ'' consisting of the following transitions:

$$\begin{array}{lll} \mathsf{a} \to (1,1) & y \to (2,2) & \mathsf{h}((1,1),(2,2)) \to (3,5) \\ & y \to (2,3) & \mathsf{h}((1,1),(2,3)) \to (3,5) \end{array}$$

Because $t \to_{\Delta''} \mathsf{h}(\mathsf{a}, (2, 2)) \to^*_{\Delta''} (3, 5)$ we conclude that $t\sigma \to^*_\Delta 5$ if we take $\sigma = \{y \mapsto 2\}$. Similarly, the sequence $t \to_{\Delta''} \mathsf{h}(\mathsf{a}, (2, 3)) \to^*_{\Delta''} (3, 5)$ gives rise to the state substitution $\tau = \{y \mapsto 3\}$ with $t\tau \to^*_\Delta 5$.

It is not difficult to observe that this approach has two major advantages over the matching algorithm proposed in the previous subsection. First of all, all state substitutions σ such that $t\sigma \to^*_\Delta q$ for some $q \in Q$ are computed independently from the actual state q. In case of the matching algorithm the situation is different. There, the solutions for each matching problem $t \vdash q$ with $q \in Q$ are computed separately. Secondly, the approach based on tree automata uses implicitly sharing of terms to avoid expensive recalculations. The matching algorithm is not equipped with such a feature. So it can easily happen that solutions of intermediate matching problems are computed several times. As an immediate consequence, the average complexity of the matching algorithm is worse than the average complexity of the approach based on tree automata techniques.

Of course, using tree automata to find compatibility violations also possesses some disadvantages. The most serious one is that we have to enumerate all accepting sequences to get the substitutions we are looking for. So compared to the matching algorithm, where we can directly read off all possible substitutions, the approach based on tree automata somehow encodes the computed substitutions.

3.3 Solving Compatibility Violations

To obtain a compatible tree automaton we have to solve violations of the compatibility requirement by establishing unavailable paths. Since there are various ways to do this, in [14] so called *abstraction functions* have been introduced to characterize possible approximation techniques. Below we recall the basic definitions.

Let $\mathcal{F}$ be a signature and Q a set of states. An *abstraction function* is a mapping ϕ which assigns to any term $f(q_1,\dots,q_n)$ a state $q \in Q$. Here $f \in \mathcal{F}$ is a n-ary function symbol and $q_1,\dots,q_n \in Q$. We extend abstraction functions to ground terms over the extended signature $\mathcal{F} \cup Q$. For a term $t \in \mathcal{T}(\mathcal{F} \cup Q)$ the function $\phi(t)$ is defined as follows:

$$\phi(t) = \begin{cases} t & \text{if } t \in Q \\ \phi(f(\phi(t_1),\dots,\phi(t_n))) & \text{if } t = f(t_1,\dots,t_n) \end{cases}$$

Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a tree automaton, $t \in \mathcal{T}(\mathcal{F} \cup Q)$ a term, and $q \in Q$ a state. To establish a path $t \to^* q$ we use a function $\mathsf{norm}_\phi(t,q)$ which computes a set of transitions using an abstraction function ϕ such that $t \to^*_{\mathsf{norm}_\phi(t,q)} q$. Here $\mathsf{norm}_\phi(t,q)$ is defined as $\mathsf{norm}_\phi(t,q) = \varnothing$ if $t = q$ and $\mathsf{norm}_\phi(t,q) = \{t \to q\}$ if $t \in Q$ and $t \neq q$. If $t = f(t_1,\dots,t_n)$ then the set $\mathsf{norm}_\phi(t,q)$ consists of the transition $f(\phi(t_1),\dots,\phi(t_n)) \to q$ as well as all transitions contained in the sets $\mathsf{norm}_\phi(t_i,\phi(t_i))$ with $i \in \{1,\dots,n\}$. It is not difficult to see that by adding all transitions in $\mathsf{norm}_\phi(t,q)$ to Δ we have $t \to^*_\Delta q$.

Example 3.6. Let $\mathcal{R}$ be the TRS and $\mathcal{A}$ the tree automaton of Example 3.2. In the following we construct a tree automaton that is compatible with $\mathcal{R}$ and $L = \mathcal{L}(\mathcal{A})$ by using an abstraction function ϕ defined as follows:

$$\phi(\mathsf{g}(1)) = 4 \qquad \phi(\mathsf{g}(2)) = 6 \qquad \phi(\mathsf{g}(3)) = 7 \qquad \phi(\mathsf{g}(4)) = 4$$

We start by solving the compatibility violation caused by the rewrite rule $\mathsf{f}(x,y) \to_\mathcal{R} \mathsf{f}(\mathsf{g}(x),\mathsf{g}(y))$. We have $\mathsf{f}(4,1) \to 5$ but not $\mathsf{f}(\mathsf{g}(4),\mathsf{g}(1)) \to^* 5$. Since $\mathsf{norm}_\phi(\mathsf{f}(\mathsf{g}(4),\mathsf{g}(1)),5) = \{\mathsf{g}(1) \to 4, \mathsf{g}(4) \to 4, \mathsf{f}(4,4) \to 5\}$ we establish the path $\mathsf{f}(\mathsf{g}(4),\mathsf{g}(1)) \to^* 5$ by adding the transition $\mathsf{f}(4,4) \to 5$. Note that the other two transitions do not have to be added since they are already present. Next we consider the compatibility violation $\mathsf{h}(\mathsf{a},2) \to^* 5$ and $\mathsf{h}(\mathsf{g}(2),\mathsf{g}(2)) \not\to^* 5$ caused by the rewrite rule $\mathsf{h}(\mathsf{a},y) \to_\mathcal{R} \mathsf{g}(\mathsf{g}(y),\mathsf{g}(y))$. Since $\phi(\mathsf{g}(2)) = 6$ we have $\mathsf{norm}_\phi(\mathsf{h}(\mathsf{g}(2),\mathsf{g}(2)),5) = \{\mathsf{g}(2) \to 6, \mathsf{h}(6,6) \to 5\}$. Hence we add the fresh state 6 and the new transitions $\mathsf{g}(2) \to 6$ and $\mathsf{h}(6,6) \to 5$. Finally the violation caused by the rewrite rule $\mathsf{h}(\mathsf{a},y) \to_\mathcal{R} \mathsf{g}(\mathsf{g}(y),\mathsf{g}(y))$ with $\mathsf{h}(\mathsf{a},3) \to^* 5$ is solved by adding the new state 7 and the transitions $\mathsf{g}(3) \to 7$ and $\mathsf{h}(7,7) \to 5$ to establish $\mathsf{h}(\mathsf{g}(3),\mathsf{g}(3)) \to^* 5$. Here $\mathsf{norm}_\phi(\mathsf{h}(\mathsf{g}(3),\mathsf{g}(3)),5) = \{\mathsf{g}(3) \to 7, \mathsf{h}(7,7) \to 5\}$. After this step the constructed tree automaton is compatible with $\mathcal{R}$ and L. It is not difficult to observe that the language accepted by the constructed tree automaton represents a quite optimal regular over-approximation of $\to^*_\mathcal{R}(L)$. So $\mathcal{L}(\mathcal{A})$ consists of $\mathsf{h}(\mathsf{a},\mathsf{b})$, $\mathsf{h}(\mathsf{a},\mathsf{c})$, $\mathsf{h}(\mathsf{g}(\mathsf{b}),\mathsf{g}(\mathsf{b}))$, and $\mathsf{h}(\mathsf{g}(\mathsf{c}),\mathsf{g}(\mathsf{c}))$ as well as all terms of the form $\mathsf{f}(\mathsf{g}^*(\mathsf{a}),\mathsf{g}^*(\mathsf{a}))$.

It is easy to see that we always obtain a compatible tree automaton if the domain of the used abstraction function ϕ is finite. For some restricted classes of TRSs, like ground TRSs, it is even known which (finite) abstraction functions produce optimal over-approximations [14]. However, in general it is unclear how to automatically construct suitable abstraction functions. Therefore several approximation techniques have been developed. In [14] combinations of injective abstraction functions and so called *approximation equations* are used to construct a compatible tree automaton. Alternatively one can follow the approach in [20] and use *approximation rules* to solve violations of the compatibility requirement. Another common method is to use *specialized* approximation functions which reuse transitions of the underlying tree automaton to establish missing paths [23, 24, 63]. In the following we will shortly introduce and compare all three methods.

3.3.1 Approximation Equations

Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a tree automaton. An *approximation equation* is an equation $s = t$ where $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. For two states $p, q \in Q$ the function $\mathsf{merge}(\mathcal{A}, p, q) = (\mathcal{F}, Q', Q'_f, \Delta')$ denotes the tree automaton that is obtained from $\mathcal{A}$ by replacing state p by q. That is $Q' = (Q \setminus \{p\}) \cup \{q\}$ and $Q'_f = Q_f$ if $p \notin Q_f$ and $Q'_f = (Q_f \setminus \{p\}) \cup \{q\}$ otherwise. The set Δ' is obtained from Δ by replacing state p by q in all left- and right-hand sides of transitions in Δ. Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a tree automaton, $t \in \mathcal{T}(\mathcal{F} \cup Q,)$ a term, and $q \in Q$ a state. To establish a path $t \to^* q$ using an injective abstraction function ϕ and a set of approximation rules $\mathcal{E}$ we perform the following steps. At first we establish the path $t \to^* q$ by adding the transitions $\mathsf{norm}_\phi(t, q)$ to Δ. After that we apply the function $\mathsf{merge}(\mathcal{A}, p_s, p_t)$ as long as there is an equation $s = t \in \mathcal{E}$ and a state substitution σ such that $s\sigma \to^* p_s$ and $t\sigma \to^* p_t$.

Example 3.7. Let us consider again the TRS $\mathcal{R}$ and the initial tree automaton $\mathcal{A}$ of Example 3.2. In the following we construct a compatible tree automaton using the injective abstraction function ϕ defined as $\phi(\mathsf{g}(2)) = 6$ and $\phi(\mathsf{g}(3)) = 7$ as well as the approximation equation $\mathsf{g}(x) = x$. We start by adapting $\mathcal{A}$ such that it fulfills the constraints induced by the used approximation rule. Since $\mathsf{g}(x)\sigma \to 4$ and $x\sigma = 1$ with $\sigma = \{x \mapsto 1\}$ we replace 4 by 1 using the function $\mathsf{merge}(\mathcal{A}, 4, 1)$. Thereby we obtain a tree automaton consisting of the following transitions:

$$\begin{array}{lll} \mathsf{a} \to 1 & \mathsf{g}(1) \to 1 & \mathsf{f}(1,1) \to 5 \\ \mathsf{b} \to 2 & & \mathsf{h}(1,2) \to 5 \\ \mathsf{c} \to 3 & & \mathsf{h}(1,3) \to 5 \end{array}$$

Next we solve the violation $\mathsf{h}(\mathsf{a}, 2) \to^* 5$ but not $\mathsf{h}(\mathsf{g}(2), \mathsf{g}(2)) \to^* 5$ caused by the rewrite rule $\mathsf{h}(\mathsf{a}, y) \to_{\mathcal{R}} \mathsf{h}(\mathsf{g}(y), \mathsf{g}(y))$. According to the abstraction function ϕ we establish the path $\mathsf{h}(\mathsf{g}(2), \mathsf{g}(2)) \to^* 5$ by adding the transitions $\mathsf{g}(2) \to 6$ and $\mathsf{h}(6, 6) \to 5$. After that we merge state 6 and 2 because $\mathsf{g}(x)\sigma \to 6$ and $x\sigma = 2$ with $\sigma = \{x \mapsto 2\}$. Hence we replace the transitions $\mathsf{g}(2) \to 6$ and

$\mathsf{h}(6,6) \to 5$ by $\mathsf{g}(2) \to 2$ and $\mathsf{h}(2,2) \to 5$. Similarly, the compatibility violation $\mathsf{h}(\mathsf{a},3) \to^* 5$ but not $\mathsf{h}(\mathsf{g}(3),\mathsf{g}(3)) \to^* 5$, caused by the rewrite rule $\mathsf{h}(\mathsf{a},y) \to_{\mathcal{R}} \mathsf{h}(\mathsf{g}(y),\mathsf{g}(y))$, is solved by adding the transitions $\mathsf{g}(3) \to 3$ and $\mathsf{h}(3,3) \to 5$. Note that the abstraction function ϕ yields the transitions $\mathsf{g}(3) \to 7$ and $\mathsf{h}(7,7) \to 5$ which are then replaced by $\mathsf{g}(3) \to 3$ and $\mathsf{h}(3,3) \to 5$ using the function $\mathsf{merge}(\mathcal{A},7,3)$. After this step the constructed tree automaton is compatible with $\mathcal{R}$ and accepts all terms of the form $\mathsf{f}(\mathsf{g}^*(\mathsf{a}),\mathsf{g}^*(\mathsf{a}))$, $\mathsf{h}(\mathsf{g}^*(\mathsf{a}),\mathsf{g}^*(\mathsf{b}))$, $\mathsf{h}(\mathsf{g}^*(\mathsf{a}),\mathsf{g}^*(\mathsf{c}))$, $\mathsf{h}(\mathsf{g}^*(\mathsf{b}),\mathsf{g}^*(\mathsf{b}))$, and $\mathsf{h}(\mathsf{g}^*(\mathsf{c}),\mathsf{g}^*(\mathsf{c}))$.

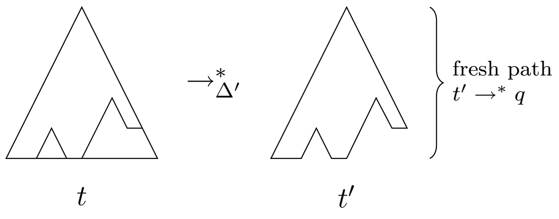
3.3.2 Approximation Rules

The approach base on approximation rules is very different from the previous one. At first we do not use abstraction functions to construct a missing path $t \to^* q$. Instead of that we use a set of rewrite rules $\mathcal{S}$ to rewrite the term t to some smaller term t' by contracting innermost redexes only. Thereby, each rewrite rule $l \to r \in \mathcal{S}$ fulfills the property that if the variables of l and r are instantiated by states, the resulting rule equates to an ordinary transition. So by adding all transitions applied in the derivation $t \to_{\mathcal{S}}^* t'$ to the underlying tree automaton we know that t can be reduced to t'. Since it might be the case that $t' \neq q$ we still have to establish the path $t' \to^* q$ to ensure that $t \to_{\mathcal{S}}^* t' \to^* q$. So, in contrast to the first approach it can happen that after the application of an approximation rule, it still does not hold that $t \to^* q$. To establish $t' \to^* q$ we use an injective abstraction function which assigns to each proper subterm of t' a fresh state.

A big advantage of this approach is that it sometimes allows us to obtain more precise approximations because it takes the structure of the underlying tree automaton into account. However, one has to be very careful during the specification of approximation rules because it can easily happen that new states and transitions are kept being added and hence that the completion procedure does not terminate. Below we formally introduce this approach.

Let $\mathcal{F}$ be a signature and Q a set of states. An *approximation rule* is a pair $(l \to x, \mathcal{S})$ where $l \in \mathcal{T}(\mathcal{F} \cup Q, \mathcal{V})$ is a term, $x \in \mathcal{V} \cup Q$ a variable or a state, and $\mathcal{S} = \{l_1 \to x_1, \ldots, l_n \to x_n\}$ a set of rewrite rules such that $\mathsf{depth}(l_i) \leqslant 1$, $l_i \in \mathcal{T}(\mathcal{F} \cup Q, \mathcal{V})$, and $x_i \in \mathcal{V}\mathsf{ar}(l_i) \cup \mathcal{V}\mathsf{ar}(l) \cup \{x\} \cup Q$ for all $i \in \{1, \ldots, n\}$. Similar as in the first approach we use a function $\mathsf{norm}_{(l \to x, \mathcal{S})}(t,q)$ to define the application of an approximation rule $(l \to x, \mathcal{S})$ to a ground term t over the signature $\mathcal{F} \cup Q$ and a state $q \in Q$. Let $\Delta_{(l \to x, \mathcal{S})}(t,q)$ denote the set of all transitions $u \to p$ such that there are a rewrite rule $l' \to x' \in \mathcal{S}$ and a state substitution $\sigma\colon \mathcal{V} \to Q$ with $l\sigma = t$, $x\sigma = q$, $u = l'\sigma$, and $p = x'\sigma$. We have $(t', \Delta') \in \mathsf{norm}_{(l \to x, \mathcal{S})}(t,q)$ for all ground terms t' and $\Delta' \subseteq \Delta_{(l \to x, \mathcal{S})}(t,q)$ such that $t \to_{\Delta'}^* t'$ where each transition in Δ' is applied at least once. Note that it always holds that $(t, \varnothing) \in \mathsf{norm}_{(l \to x, \mathcal{S})}(t,q)$. Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a tree automaton, $t \in \mathcal{T}(\mathcal{F} \cup Q)$ a term, and $q \in Q$ a state. Let ψ be a set of approximation rules. To establish a path $t \to^* q$ we precede as illustrated in Figure 3.2. At first we choose an approximation rule $(l \to x, \mathcal{S}) \in \psi$ such that $t \to_{\Delta'}^* t'$ with $(t', \Delta') \in \mathsf{norm}_{(l \to x, \mathcal{S})}(t,q)$ and $|t'|$ is minimal. After that we add all transitions of Δ' to Δ. Furthermore, if $t' \neq q$ we add some new transitions

Figure 3.2: Approximation rules



involving new states to ensure that $t' \to^* q$.

Example 3.8. Assume that we want to construct a tree automaton that is compatible with the TRS $\mathcal{R}$ and the language $\mathcal{L}(\mathcal{A})$ accepted by the tree automaton $\mathcal{A}$ of Example 3.2, using the following approximation rules:

$$(\mathsf{f}(\mathsf{g}(x), \mathsf{g}(y)) \to z, \{\mathsf{g}(x) \to 4, \mathsf{g}(y) \to 4, \mathsf{f}(4,4) \to z\})$$
$$(\mathsf{h}(\mathsf{g}(y), \mathsf{g}(y)) \to z, \{\mathsf{g}(y) \to 6, \mathsf{h}(6,6) \to z\})$$

The first violation that we consider, $\mathsf{f}(4,1) \to 5$ and $\mathsf{f}(\mathsf{g}(4),\mathsf{g}(1)) \not\to^* 5$, is caused by the rewrite rule $\mathsf{f}(x,y) \to_{\mathcal{R}} \mathsf{f}(\mathsf{g}(x),\mathsf{g}(y))$. To establish the missing path we use the first approximation rule which yields the new transition $\mathsf{f}(4,4) \to 5$. Next we solve the compatibility violations caused by the rewrite rule $\mathsf{h}(\mathsf{a},y) \to_{\mathcal{R}} \mathsf{h}(\mathsf{g}(y),\mathsf{g}(y))$. We have $\mathsf{h}(\mathsf{a},2) \to 5$ but not $\mathsf{h}(\mathsf{g}(2),\mathsf{g}(2)) \to^* 5$. According to the second approximation rule we add the new state 6 and the transitions $\mathsf{g}(2) \to 6$ and $\mathsf{h}(6,6) \to 5$. Finally, the violation $\mathsf{h}(\mathsf{a},3) \to 5$ but $\mathsf{h}(\mathsf{g}(3),\mathsf{g}(3)) \not\to^* 5$ is resolved by adding the new transition $\mathsf{g}(3) \to 6$. The resulting automaton is compatible with $\mathcal{R}$ and accepts the terms $\mathsf{h}(\mathsf{a},\mathsf{b})$, $\mathsf{h}(\mathsf{a},\mathsf{c})$, and $\mathsf{h}(s,t)$ with $s,t \in \{\mathsf{g}(\mathsf{b}),\mathsf{g}(\mathsf{c})\}$ as well as all terms of the form $\mathsf{f}(\mathsf{g}^*(\mathsf{a}),\mathsf{g}^*(\mathsf{a}))$.

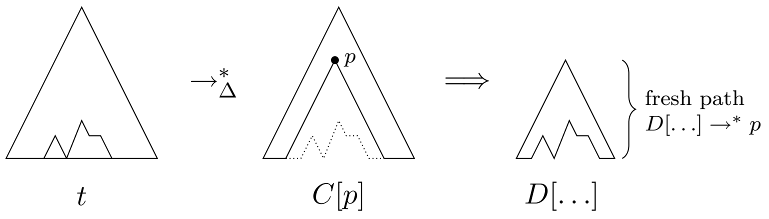
3.3.3 Approximation Functions

The third approach is quite similar to the second one. Instead of fixing the transitions that can be added to a given tree automaton we use some specialized approximation function which returns a set of transitions that have to be added to the given tree automaton in order to establish a path $t \to^* q$. The main difference between the second and the third approach is that with the latter one we can reuse more transitions of the underlying tree automaton because we can analyze the structure of the term t more precisely. By doing so we sometimes can obtain more fine grained approximations. In the following we present the approximation function introduced in [24].

Let $\mathcal{F}$ be a signature and Q a set of states. Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a tree automaton, $t \in \mathcal{T}(\mathcal{F} \cup Q)$ a term, and $q \in Q$ a state. To establish a path $t \to^* q$ the following steps are performed (see Figure 3.3):

1. Calculate all contexts $C[\square]$ and $D[\square,\ldots,\square]$ and ground terms $t_1,\ldots,t_n$ over the signature $\mathcal{F} \cup Q$ such that $C[D[t_1,\ldots,t_n]] = t$, $C[p] \to^*_\Delta q$, and $t_i \to^*_\Delta q_i$ for states $p, q, q_i \in Q$ with $i \in \{1,\ldots,n\}$.

Figure 3.3: Approximation functions



2. Choose among all possible combinations one where the size of the context $D[\Box, \ldots, \Box]$ is minimal.
3. Add new transitions involving new states to achieve $D[q_1, \ldots, q_n] \rightarrow^* p$.

Let us illustrate the above approximation function on the previous example.

Example 3.9. Similar as before we construct a tree automaton that is compatible with the TRS $\mathcal{R}$ and the language $\mathcal{L}(\mathcal{A})$ accepted by the tree automaton $\mathcal{A}$ of Example 3.2. The first compatibility violation that we consider, $\mathsf{f}(4,1) \rightarrow 5$ and $\mathsf{f}(\mathsf{g}(4), \mathsf{g}(1)) \not\rightarrow^* 5$ is caused by the rewrite rule $\mathsf{f}(x,y) \rightarrow_\mathcal{R} \mathsf{f}(\mathsf{g}(x), \mathsf{g}(y))$. According to the above algorithm we can reuse the transition $\mathsf{g}(1) \rightarrow 4$ and $\mathsf{g}(4) \rightarrow 4$ by choosing $C = \Box$, $D = \mathsf{f}(\Box, \Box)$, $t_1 = \mathsf{g}(4)$, and $t_2 = \mathsf{g}(1)$. So we just add the transition $\mathsf{f}(4,4) \rightarrow 5$. Next we consider the compatibility violations $\mathsf{h}(\mathsf{a}, 2) \rightarrow 5$ but not $\mathsf{h}(\mathsf{g}(2), \mathsf{g}(2)) \rightarrow^* 5$ caused by the rewrite rule $\mathsf{h}(\mathsf{a}, y) \rightarrow_\mathcal{R} \mathsf{h}(\mathsf{g}(y), \mathsf{g}(y))$. Since there are no transitions with left-hand side $\mathsf{g}(2)$ we have $C = \Box$ and $D = \mathsf{h}(\mathsf{g}(2), \mathsf{g}(2))$. So to establish $\mathsf{h}(\mathsf{g}(2), \mathsf{g}(2)) \rightarrow^* 5$ we add a new state 6 and transitions $\mathsf{g}(2) \rightarrow 6$ and $\mathsf{h}(6,6) \rightarrow 5$. Similarly, the violation $\mathsf{h}(\mathsf{a}, 3) \rightarrow 5$ but $\mathsf{h}(\mathsf{g}(3), \mathsf{g}(3)) \not\rightarrow^* 5$ is resolved by adding the new state 7 and the transitions $\mathsf{g}(3) \rightarrow 7$ and $\mathsf{h}(7,7) \rightarrow 5$. The resulting automaton is compatible with $\mathcal{R}$ and accepts the terms $\mathsf{h}(\mathsf{a}, \mathsf{b})$, $\mathsf{h}(\mathsf{a}, \mathsf{c})$, $\mathsf{h}(\mathsf{g}(\mathsf{b}), \mathsf{g}(\mathsf{b}))$, and $\mathsf{h}(\mathsf{g}(\mathsf{c}), \mathsf{g}(\mathsf{c}))$ as well as all terms of the form $\mathsf{f}(\mathsf{g}^*(\mathsf{a}), \mathsf{g}^*(\mathsf{a}))$. It is not difficult to check that this over-approximation is as optimal as the one constructed in Example 3.6.

3.4 Quasi-Deterministic Tree Automata

To construct the set $\rightarrow^*_\mathcal{R}(L)$ for a non-left-linear TRS $\mathcal{R}$ and a regular language L we want to use compatible tree automata as presented in Section 3.1. However there is one problem. To cope with non-left-linear TRSs, non-deterministic tree automata cannot be used [14]. The reason is that given a non-deterministic tree automaton it is possible that terms can only be rewritten by reducing equivalent subterms to different states. In [20] this problem has been solved by choosing appropriate approximation rules which keep critical transitions *deterministic* during the construction of a compatible tree automaton. A serious disadvantage of this approach is that the obtained approximations are not as exact as they could be if the usage of unrestricted approximation functions would be allowed. In order to avoid such limitations one could follow the common approach to handle non-linearity by using *deterministic* tree automata (compare [6, 48, 51]). However, in general deterministic tree automata are not suitable for tree automata completion because during the determinisation transitions might be removed which were added in earlier stages to ensure compatibility. In the following we present an approach based on so called *quasi-deterministic* tree automata. To simplify the presentation we consider tree automata without ϵ-transitions.

Definition 3.10. Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a tree automaton. We say that a state p *subsumes* a state q with $p, q \in Q$ if p is final when q is final and for all left-hand sides $f(q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n) \in \mathsf{lhs}(\Delta)$, the left-hand side $f(q_1, \ldots, q_{i-1}, p, q_{i+1}, \ldots, q_n)$ belongs to $\mathsf{lhs}(\Delta)$. For a left-hand side $l \in \mathsf{lhs}(\Delta)$, the set $\{q \mid l \to q \in \Delta\}$ of possible right-hand sides of l is denoted by $Q_\mathcal{A}(l)$. Let $\phi_\mathcal{A}$ denote a function which maps each left-hand side $l \in \mathsf{lhs}(\Delta)$ to a state $q \in Q_\mathcal{A}(l)$. The relation $\succeq_{\phi_\mathcal{A}}$ is defined as the smallest transitive relation on Q such that $\phi_\mathcal{A}(l) \succeq_{\phi_\mathcal{A}} q$ for all $l \in \mathsf{lhs}(\Delta)$ and $q \in Q_\mathcal{A}(l)$, and $\phi_\mathcal{A}(l) \succeq_{\phi_\mathcal{A}} \phi_\mathcal{A}(l')$ for all left-hand sides $l = f(p_1, \ldots, p_{i-1}, p, p_{i+1}, \ldots, p_n)$ and $l' = f(p_1, \ldots, p_{i-1}, q, p_{i+1}, \ldots, p_n)$ in $\mathsf{lhs}(\Delta)$ with $p \succeq_{\phi_\mathcal{A}} q$. The tree automaton $\mathcal{A}$ is said to be *quasi-deterministic* if there exits a function $\phi_\mathcal{A}$ such that p subsumes q whenever $p \succeq_{\phi_\mathcal{A}} q$ for all $p, q \in Q$.[2]

Deterministic tree automata are trivially quasi-deterministic because $Q_\mathcal{A}(l)$ is a singleton set for every left-hand side $l \in \mathsf{lhs}(\Delta)$. In general, Δ may admit more than one function $\phi_\mathcal{A}$ that satisfies the above property. In the following we assume that for each quasi-deterministic tree automaton, $\phi_\mathcal{A}$ denotes a fixed function that fulfills the requirements of Definition 3.10. The set of all states $\phi_\mathcal{A}(l)$ with $l \in \mathsf{lhs}(\Delta)$ is denoted by $Q_{\phi_\mathcal{A}}$ and the restriction of Δ to transitions $l \to q$ that satisfy $q = \phi_\mathcal{A}(l)$ is denoted by $\Delta_{\phi_\mathcal{A}}$. To simplify the notion we sometimes call $\phi_\mathcal{A}(l)$ the *designated state* for l and drop in the following the subscript $\mathcal{A}$ from $\phi_\mathcal{A}$ and $Q_\mathcal{A}(l)$ when no confusion can arise.

Example 3.11. The tree automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ over the signature $\mathcal{F} = \{\mathsf{a}, \mathsf{f}\}$ with $Q = \{1, 2\}$, $Q_f = \{1\}$, and the transitions

$$\mathsf{a} \to 1 \mid 2 \qquad\qquad \mathsf{f}(1,2) \to 1$$

is not quasi-deterministic. This is due to the fact that for the left-hand side a neither 2 nor 1 can be used as designated state. If we take $\phi(\mathsf{a}) = 1$ then we should be able to replace state 2 in the transition $\mathsf{f}(1,2) \to 1$ by 1, that is, the transition $\mathsf{f}(1,1) \to 1$ should belong to Δ. Similarly, if we take $\phi(\mathsf{a}) = 2$ then the transition $\mathsf{f}(2,2) \to 1$ should belong to Δ.

The key feature of a quasi-deterministic tree automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ is that it accepts the same language as $\mathcal{A}_\phi = (\mathcal{F}, Q, Q_f, \Delta_\phi)$. To prove this, we need the following result.

[2]Note that the definition of quasi-deterministic tree automata used in [43] requires that for two left-hand sides $l = f(q_1, \ldots, q_{i-1}, p, q_{i+1}, \ldots, q_n)$ and $l' = f(q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n)$ in $\mathsf{lhs}(\Delta)$ we have $Q_\mathcal{A}(l) \supseteq Q_\mathcal{A}(l')$ whenever $p \succeq_{\phi_\mathcal{A}} q$. So compared to the new definition, which just demands the existence of the single transition $l' \to \phi(l')$, the definition in [43] is more restrictive. Furthermore, as a byproduct of the new definition we have to add less transitions to make a tree automaton quasi-deterministic which in turn has a positive impact on the completion process.

Lemma 3.12. *Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a quasi-deterministic tree automaton, $t \in \mathcal{T}(\mathcal{F})$ a term, and $p \in Q$ a state. If $t \to_\Delta^* p$ then there exists a state $q \in Q$ such that $t \to_{\Delta_\phi}^* q$ and $q \succeq_\phi p$.*

Proof. We perform induction on t. If t is a constant the claim holds trivially. Let $t = f(t_1, \ldots, t_n)$. The sequence from the term t to the state p can be written as $t \to_\Delta^* f(p_1, \ldots, p_n) \to_\Delta p$. The induction hypothesis yields for every subterm t_i with $i \in \{1, \ldots, n\}$ a state $q_i \in Q$ such that $t_i \to_{\Delta_\phi}^* q_i$ and $q_i \succeq_\phi p_i$. Let $l = f(p_1, \ldots, p_n)$ and $l_i = f(q_1, \ldots, q_i, p_{i+1}, \ldots, p_n)$ for all $i \in \{1, \ldots, n\}$. Since $\mathcal{A}$ is quasi-deterministic we know by Definition 3.10 that $\phi(l) \succeq_\phi p$. Furthermore we have $l_1 \in \mathsf{lhs}(\Delta)$ because $q_1 \succeq_\phi p_1$ and hence q_1 subsumes p_1. From the definition of $\succeq_\phi$ it follows that $\phi(l_1) \succeq_\phi \phi(l)$. Because $\phi(l_1) \succeq_\phi \phi(l)$ and $\phi(l) \succeq_\phi p$ we know that $\phi(l_1) \succeq_\phi p$ by the transitivity of $\succeq_\phi$. Repeating this argument $n - 1$ times yields that the left-hand side l_n belongs to $\mathsf{lhs}(\Delta)$ and $\phi(l_n) \succeq_\phi p$. So by taking $q = \phi(l_n)$ we obtain a state $q \in Q$ such that $t \to_{\Delta_\phi}^* l_n \to_{\Delta_\phi} q$ and $q \succeq_\phi p$. □

Lemma 3.13. *Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a quasi-deterministic tree automaton. The tree automaton $\mathcal{A}_\phi = (\mathcal{F}, Q, Q_f, \Delta_\phi)$ is deterministic and $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_\phi)$.*

Proof. From the definition of Δ_ϕ it is obvious that $\mathcal{A}_\phi$ is deterministic. Furthermore, because $\Delta_\phi \subseteq \Delta$ the inclusion $\mathcal{L}(\mathcal{A}_\phi) \subseteq \mathcal{L}(\mathcal{A})$ is trivial. In order to show the reverse inclusion, let $t \in \mathcal{L}(\mathcal{A})$. So $t \to_\Delta^* p$ for some $p \in Q_f$. The previous lemma yields a $q \in Q$ such that $t \to_{\Delta_\phi}^* q$ and $q \succeq_\phi p$. Since $\mathcal{A}$ is quasi-deterministic we know that q subsumes p. Together with the fact that p is a final state we conclude that $q \in Q_f$ and thus $t \in \mathcal{L}(\mathcal{A}_\phi)$. □

Because we will use quasi-deterministic tree automata rather than non-deterministic tree automata to construct $\to_\mathcal{R}^*(L)$, we adapt the definition of compatible tree automata to make it more suitable for our purpose.

Definition 3.14. Let $\mathcal{R}$ be a non-left-linear TRS and L a language. Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a quasi-deterministic tree automaton. We say that $\mathcal{A}$ is *compatible* with $\mathcal{R}$ and L if $L \subseteq \mathcal{L}(\mathcal{A})$ and for each rewrite rule $l \to r \in \mathcal{R}$ and state substitution $\sigma \colon \mathcal{V}\mathsf{ar}(l) \to Q_\phi$ such that $l\sigma \to_{\Delta_\phi}^* q$ it holds that $r\sigma \to_\Delta^* q$.

The reason for requiring $r\sigma \to_\Delta^* q$ rather than $r\sigma \to_{\Delta_\phi}^* q$ in the above definition is that it is easier to construct a path $r\sigma \to_\Delta^* q$ because one can reuse more transitions. Besides that, one of our primary objectives is to develop an approach where arbitrary approximation techniques can be used to achieve a compatible tree automaton. If we would require $r\sigma \to_{\Delta_\phi}^* q$ we would clearly miss this target.

The general procedure to construct a quasi-deterministic tree automaton that is compatible with a non-left-linear TRS $\mathcal{R}$ and some language L is similar to the one explained in Section 3.1. Starting with some deterministic initial tree automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ which accepts L we look for violations of the compatibility requirement (recall that ϕ is uniquely determined for deterministic tree automata): $l\sigma \to_{\Delta_\phi}^* q$ and $r\sigma \not\to_\Delta^* q$ for some rewrite rule $l \to r \in \mathcal{R}$, state substitution $\sigma \colon \mathcal{V}\mathsf{ar}(l) \to Q_\phi$, and state $q \in Q_\phi$. Then we add new states and transitions to the current automaton to establish $r\sigma \to_\Delta^* q$. After that we check if the new automaton is quasi-deterministic. If this is not the case, we transform it into a tree automaton which has this property. How this can be automatically done is explained in Section 3.5. Afterwards, we search for further violations of the compatibility requirement. This process is repeated until a compatible and quasi-deterministic tree automaton is obtained.

Now, assume that we have constructed a quasi-deterministic tree automaton $\mathcal{A}$ that is compatible with a TRS $\mathcal{R}$ and a language L. To infer that $\mathcal{A}$ can be used for reachability analysis, it must be guaranteed that $\mathcal{A}$ accepts at least the set $\to_\mathcal{R}^*(L)$. To prove this main result, we need the following technical lemma.

Lemma 3.15. *Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a quasi-deterministic tree automaton and C a ground context over the signature $\mathcal{F}$. Let $p, q \in Q$ be two states such that $q \succeq_\phi p$. If $C[p] \to_{\Delta_\phi}^* p'$ for some $p' \in Q$ then there is a state $q' \in Q$ such that $C[q] \to_{\Delta_\phi}^* q'$ and $q' \succeq_\phi p'$.*

Proof. We use induction on C. If $C = \Box$ the claim holds trivially. Let $C[p] = f(t_1, \dots, t_{i-1}, D[p], t_{i+1}, \dots, t_n)$. The sequence from the term $C[p]$ to the state p' can be written as $C[p] \to_{\Delta_\phi}^* f(p_1, \dots, p_n) \to_{\Delta_\phi} p'$. The induction hypothesis yields a state $q_i \in Q$ such that $D[q] \to_{\Delta_\phi}^* q_i$ and $q_i \succeq_\phi p_i$. Because $\mathcal{A}$ is quasi-deterministic we know that q_i subsumes p_i. Hence, the left-hand side $l = f(p_1, \dots, p_{i-1}, q_i, p_{i+1}, \dots, p_n)$ belongs to $\mathsf{lhs}(\Delta)$. Because $q_i \succeq_\phi p_i$ and $\phi(f(p_1, \dots, p_n)) = p'$ we have $\phi(l) \succeq_\phi p'$. So by taking $q' = \phi(l)$ we obtain $C[q] \to_{\Delta_\phi}^* l \to_{\Delta_\phi} q'$ and $q' \succeq_\phi p'$ as desired. $\Box$

Theorem 3.16. *Let $\mathcal{R}$ be a TRS, L a language, and $\mathcal{A}$ a quasi-deterministic tree automaton. If $\mathcal{A}$ is compatible with $\mathcal{R}$ and L then $\to_\mathcal{R}^*(L) \subseteq \mathcal{L}(\mathcal{A})$.*

Proof. Let s and t be two ground terms such that $s \in \mathcal{L}(\mathcal{A})$ and $s \to_\mathcal{R} t$. We show that $t \in \mathcal{L}(\mathcal{A})$. The desired result then follows by induction. Since $s \to_\mathcal{R} t$ there exist a rewrite rule $l \to r \in \mathcal{R}$, a context C, and a ground substitution σ such that $s = C[l\sigma] \to_\mathcal{R} C[r\sigma] = t$. Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ and $\mathcal{A}_\phi = (\mathcal{F}, Q, Q_f, \Delta_\phi)$. Because $s \in \mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_\phi)$ according to Lemma 3.13, there exist states $p \in Q$ and $p' \in Q_f$ such that $s = C[l\sigma] \to_{\Delta_\phi}^* C[p] \to_{\Delta_\phi}^* p'$. Because $\mathcal{A}_\phi$ is deterministic by Lemma 3.13, different occurrences of $x\sigma$ in $l\sigma$ with $x \in \mathcal{V}\mathsf{ar}(l)$ are reduced to the same state in the sequence from $C[l\sigma]$ to $C[p]$. Hence there exists a mapping $\tau\colon \mathcal{V}\mathsf{ar}(l) \to Q_\phi$ such that $l\sigma \to_{\Delta_\phi}^* l\tau \to_{\Delta_\phi}^* p$. Compatibility of $\mathcal{A}$ yields $r\sigma \to_\Delta^* p$. By applying Lemma 3.12 we obtain a state $q \in Q$ such that $r\sigma \to_{\Delta_\phi}^* r\tau \to_{\Delta_\phi}^* q$ and $q \succeq_\phi p$. Because $C[p] \to_{\Delta_\phi}^* p'$ and $q \succeq_\phi p$ we know from Lemma 3.15 that there is a state $q' \in Q$ such that $C[q] \to_{\Delta_\phi}^* q'$ and $q' \succeq_\phi p'$. Since $\mathcal{A}$ is quasi-deterministic we know that q' subsumes p' and hence that $q' \in Q_f$. So $t = C[r\sigma] \to_{\Delta_\phi}^* C[q] \to_{\Delta_\phi}^* q'$ and thus $t \in \mathcal{L}(\mathcal{A})$. $\Box$

The reason why we prefer quasi-deterministic tree automata over deterministic automata is the importance of preserving existing transitions when constructing an automaton that satisfies the compatibility condition. This is illustrated in the next example.

Example 3.17. Consider the TRS $\mathcal{R}$ consisting of the rewrite rules

$$\mathsf{f}(x,x) \to \mathsf{f}(\mathsf{a},\mathsf{b}) \qquad\qquad \mathsf{f}(\mathsf{a},\mathsf{a}) \to \mathsf{a}$$

and the initial tree automaton consisting of the transitions $\mathsf{a} \to 1$ and $\mathsf{f}(1,1) \to 2$ as well as the final state 2, accepting the language $L = \{\mathsf{f}(\mathsf{a},\mathsf{a})\}$. Suppose we look for a deterministic tree automaton that is compatible with $\mathcal{R}$ and L. Since $\mathsf{f}(\mathsf{a},\mathsf{a}) \to_{\mathcal{R}} \mathsf{a}$ and $\mathsf{f}(\mathsf{a},\mathsf{a}) \to^* 2$, we add the transition $\mathsf{a} \to 2$. Next we consider $\mathsf{f}(x,x) \to_{\mathcal{R}} \mathsf{f}(\mathsf{a},\mathsf{b})$ with $\mathsf{f}(1,1) \to 2$. In order to ensure $\mathsf{f}(\mathsf{a},\mathsf{b}) \to^* 2$ we may reuse one of the transitions $\mathsf{a} \to 1$ and $\mathsf{a} \to 2$. Let us consider the various alternatives.

- Suppose we reuse the transition $\mathsf{a} \to 1$. Then we can solve the existing compatibility violation by adding a new state 3 and transitions $\mathsf{b} \to 3$ and $\mathsf{f}(1,3) \to 2$, resulting in the following transitions:

  $$\mathsf{a} \to 1 \mid 2 \qquad \mathsf{f}(1,1) \to 2 \qquad \mathsf{b} \to 3 \qquad \mathsf{f}(1,3) \to 2$$

  Making these transitions deterministic produces an automaton that includes $\mathsf{a} \to \{1,2\}$ and $\mathsf{f}(1,1) \to 2$. To simplify the presentation we identify states $\{q\}$ with $q \in Q$ by q. Because the transition $\mathsf{a} \to 2$ was removed, the first violation of compatibility that we considered, $\mathsf{f}(\mathsf{a},\mathsf{a}) \to_{\mathcal{R}} \mathsf{a}$ and $\mathsf{f}(\mathsf{a},\mathsf{a}) \to^* 2$, reappears. So we have to add $\mathsf{a} \to 2$ again, but each time we make the automaton deterministic this transition is deleted.

- The remaining options would be to reuse the transition $\mathsf{a} \to 2$ or to choose a fresh state for a. However they all give rise to a similar situation as before.

So by using deterministic automata we will never achieve compatibility. The problem is clearly the removal of transitions that were added in an earlier stage to ensure compatibility and that is precisely the reason why we introduced quasi-deterministic tree automata. Starting from the transitions in the last case above, the following quasi-deterministic tree automaton is constructed:

$$\begin{array}{cccc} \mathsf{a} \to 1 \mid 2 \mid 4 & \mathsf{b} \to 3 & \mathsf{f}(1,1) \to 2 & \mathsf{f}(1,3) \to 2 \\ \mathsf{f}(4,1) \to 2 & \mathsf{f}(1,4) \to 2 & \mathsf{f}(4,4) \to 2 & \mathsf{f}(4,3) \to 2 \end{array}$$

The path $\mathsf{f}(\mathsf{a},\mathsf{b}) \to^* 2$ has been established by reusing the transition $\mathsf{a} \to 1$ and by adding a new state 3 and the transitions $\mathsf{b} \to 3$ and $\mathsf{f}(1,3) \to 1$ to the automaton. To satisfy the requirements of Definition 3.10 we choose $\phi(\mathsf{a}) = 4$, $\phi(\mathsf{b}) = 3$, and $\phi(l) = 2$ for all left-hand sides l with root symbol f. Thereby state 4 has only been introduced to obtain a designated state for the transitions with left-hand side a. By defining ϕ as above we have $4 \succeq_\phi 1$, $4 \succeq_\phi 2$, $2 \succeq_\phi 2$, and $3 \succeq_\phi 3$. So to ensure that the constructed tree automaton is indeed quasi-deterministic it must be guaranteed that 4 subsumes 1 and 2. This can be easily done by replacing state 1 by 4 in the left-hand sides of the transitions $\mathsf{f}(1,1) \to 2$ and $\mathsf{f}(1,3) \to 2$, yielding the transitions listed in the last row. Note that $2 \succeq_\phi 2$ and $3 \succeq_\phi 3$ can be ignored because a state always subsumes itself.

3.5 Establishing Quasi-Determinism

In this section we present two approaches that can be used to transform a given tree automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ into a quasi-deterministic one *without losing any transitions of* Δ. The first algorithm is an exact transformation based on the subset construction. The second approach constructs a quasi-deterministic tree automaton which approximates the language of the given automaton by limiting the number of states that can be added to make the given automaton quasi-deterministic. To simplify the presentation we consider tree automata without ϵ-transitions.

3.5.1 Constructing Quasi-Deterministic Tree Automata

A simple and exact procedure to turn a tree automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ into an equivalent quasi-deterministic one without losing any transitions of Δ is the following:

1. Use the subset construction to transform $\mathcal{A}$ into a deterministic tree automaton $\mathcal{A}' = (\mathcal{F}, Q', Q'_f, \Delta')$.
2. Take the union of the two tree automata $\mathcal{A}$ and $\mathcal{A}'$ after identifying states $\{q\} \in Q'$ with $q \in Q$.

Let us illustrate this algorithm on a small example.

Example 3.18. Consider the tree automaton $\mathcal{A}$ of Example 3.11. The subset construction yields a tree automaton $\mathcal{A}'$ with final states $\{1\}$ and $\{1, 2\}$ and the following transitions:

$$\begin{array}{lrr} \mathsf{a} \to \{1,2\} & \mathsf{f}(\{1\},\{2\}) \to \{1\} & \mathsf{f}(\{1\},\{1,2\}) \to \{1\} \\ & \mathsf{f}(\{1,2\},\{2\}) \to \{1\} & \mathsf{f}(\{1,2\},\{1,2\}) \to \{1\} \end{array}$$

Combining the tree automata $\mathcal{A}$ and $\mathcal{A}'$ after identifying state $\{1\}$ with 1 and state $\{2\}$ with 2 produces a tree automaton consisting of the transitions

$$\begin{array}{lrr} \mathsf{a} \to 1 \mid 2 \mid \{1,2\} & \mathsf{f}(1,2) \to 1 & \mathsf{f}(1,\{1,2\}) \to 1 \\ & \mathsf{f}(\{1,2\},2) \to 1 & \mathsf{f}(\{1,2\},\{1,2\}) \to 1 \end{array}$$

and the final states 1 and $\{1, 2\}$. Note that the designated state of the left-hand side a is $\{1, 2\}$. Furthermore the transitions $\mathsf{f}(1,\{1,2\}) \to 1$, $\mathsf{f}(\{1,2\},2) \to 1$, and $\mathsf{f}(\{1,2\},\{1,2\}) \to 1$ are added to guarantee that $\{1, 2\}$ subsumes 1 and 2.

The next theorem states that the proposed quasi-determinism procedure is correct. To simplify the proof we use the following notions. Let $\mathcal{F}$ be some signature and Q a set of states. The function $\mathsf{unif}_Q \colon Q \cup 2^Q \to 2^Q$ is defined as $\mathsf{unif}_Q(q) = \{q\}$ if $q \in Q$ and $\mathsf{unif}_Q(q) = q$ otherwise. We write $\mathsf{unif}_Q(t)$ to denote the term $f(\mathsf{unif}_Q(q_1), \ldots, \mathsf{unif}_Q(q_n))$, provided that $t = f(q_1, \ldots, q_n)$ with $f \in \mathcal{F}$ and $q_i \in Q$ for all $i \in \{1, \ldots, n\}$.

Theorem 3.19. *For every tree automaton $\mathcal{A}$ there is a quasi-deterministic tree automaton $\mathcal{A}''$ such that $\mathcal{L}(\mathcal{A}'') = \mathcal{L}(\mathcal{A})$.*

Proof. Let $\mathcal{A}'' = (\mathcal{F}, Q'', Q''_f, \Delta'')$ be the tree automaton obtained from the automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ by applying the above quasi-determinism procedure and let $\mathcal{A}' = (\mathcal{F}, Q', Q'_f, \Delta')$ be the deterministic tree automaton constructed in step one. Let unif abbreviate unif_Q and ϕ abbreviate $\phi_{\mathcal{A}''}$. First we show that $\mathcal{A}''$ is quasi-deterministic. For this purpose let $\phi(l) = q$ if $\mathsf{unif}(l) \to \{q\} \in \Delta'$ and $\phi(l) = q$ if $\mathsf{unif}(l) \to q \in \Delta'$ with $|q| > 1$, for all left-hand sides $l \in \mathsf{lhs}(\Delta'')$. Note that ϕ is uniquely and totally defined because $\mathcal{A}'$ is deterministic and for all $l \in \mathsf{lhs}(\Delta'')$ we have $\mathsf{unif}(l) \in \mathsf{lhs}(\Delta')$ by the construction of $\mathcal{A}'$. We start by proving that $\mathsf{unif}(p) \supseteq \mathsf{unif}(q)$ whenever $p \succeq_\phi q$ for all states $p, q \in Q''$. Assume that $p \succeq_\phi q$ with $p = \phi(l)$ and $q \in Q_{\mathcal{A}''}(l)$ for some $l \in \mathsf{lhs}(\Delta'')$. According to the construction of $\mathcal{A}''$ we have $\mathsf{unif}(l) \to \mathsf{unif}(q) \in \Delta'$ or $l \to q \in \Delta$. If $\mathsf{unif}(l) \to \mathsf{unif}(q) \in \Delta'$ then $\mathsf{unif}(\phi(l)) = \mathsf{unif}(q)$ by the definition of ϕ and the determinism of $\mathcal{A}'$, and hence $\mathsf{unif}(p) = \mathsf{unif}(\phi(l)) \supseteq \mathsf{unif}(q)$. Similarly, if $l \to q \in \Delta$ then $\mathsf{unif}(p) \supseteq \mathsf{unif}(q)$ by the definition of ϕ and the construction of $\mathcal{A}'$. It remains to show that $p \succeq_\phi q$ if $p = \phi(l)$ and $q = \phi(l')$ for left-hand sides $l = f(p_1, \ldots, p_{i-1}, p', p_{i+1}, \ldots, p_n)$ and $l' = f(p_1, \ldots, p_{i-1}, q', p_{i+1}, \ldots, p_n)$ in $\mathsf{lhs}(\Delta'')$ with $\mathsf{unif}(p') \supseteq \mathsf{unif}(q')$. The desired result follows then by induction and the transitivity of the relation $\supseteq$. From the construction of $\mathcal{A}''$ we know that $\mathsf{unif}(l)$ and $\mathsf{unif}(l')$ belong to $\mathsf{lhs}(\Delta')$. Let p_l and $q_{l'}$ be the corresponding right-hand sides such that $\mathsf{unif}(l) \to p_l$ and $\mathsf{unif}(l') \to q_{l'}$ in Δ'. Because $\mathsf{unif}(p') \supseteq \mathsf{unif}(q')$ it follows that $p_l \supseteq q_{l'}$ by the construction of $\mathcal{A}'$. In addition we have $\mathsf{unif}(\phi(l)) = p_l$ and $\mathsf{unif}(\phi(l')) = q_{l'}$ by the choice of ϕ. Combining both results yields $\mathsf{unif}(p) = \mathsf{unif}(\phi(l)) = p_l \supseteq q_{l'} = \mathsf{unif}(\phi(l')) = \mathsf{unif}(q)$.

According to the previous result it suffice to prove that p subsumes q with respect to Δ'' if $\mathsf{unif}(p) \supseteq \mathsf{unif}(q)$ for all states $p, q \in Q''$ in order to conclude that $\mathcal{A}''$ is quasi-deterministic. Fix p and q. Because $\mathsf{unif}(l) \in \mathsf{lhs}(\Delta')$ for all $l \in \mathsf{lhs}(\Delta'')$ it is easy to see that p subsumes q with respect to Δ'' if and only if $\mathsf{unif}(p)$ subsumes $\mathsf{unif}(q)$ with respect to Δ'. Since $Q' = 2^Q$ and $f(q_1, \ldots, q_n)$ belongs to $\mathsf{lhs}(\Delta')$ for all $f \in \mathcal{F}$ and $q_i \in Q'$ with $i \in \{1, \ldots, n\}$, we know that $\mathsf{unif}(p)$ subsumes $\mathsf{unif}(q)$ with respect to Δ'. Hence we conclude that p subsumes q with respect to Δ''. This finishes the proof that $\mathcal{A}''$ is quasi-deterministic.

It remains to show that $\mathcal{L}(\mathcal{A}'') = \mathcal{L}(\mathcal{A})$. From Lemma 3.13 we know that $\mathcal{L}(\mathcal{A}'') = \mathcal{L}(\mathcal{A}''_\phi)$ with $\mathcal{A}''_\phi = (\mathcal{F}, Q'', Q''_f, \Delta''_\phi)$. Since $\mathcal{A}'$ is identical to the tree automaton $\mathcal{A}''_\phi$ after identifying states $\{q\}$ with q we conclude that $\mathcal{L}(\mathcal{A}''_\phi) = \mathcal{L}(\mathcal{A}')$. Together with the fact that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$ due to the subset construction, we know that $\mathcal{L}(\mathcal{A}'') = \mathcal{L}(\mathcal{A})$. □

It is obvious that the presented procedure is not optimal since it completely ignores any information about existing designated states. Furthermore, each time we make a tree automaton quasi-deterministic, it might happen that the resulting tree automaton is exponentially larger than the one before. In our setting this behavior is very critical because during the completion process it often happens that the currently constructed tree automaton has to be made quasi-deterministic. Since every additional state and transition introduced by the quasi-determinisation procedure in some previous step could entail that the completion procedure fails due to time constraints, it is clear that the above procedure is not ideal. Let us illustrate the addressed behavior on an example.

Example 3.20. Let $\mathcal{A}$ be the tree automaton constructed in Example 3.18 extended by the single transition $\mathsf{f}(1,2) \to \{1,2\}$. (Assume that this transition has been added in order to solve some compatibility violation.) Clearly, $\mathcal{A}$ is no longer quasi-deterministic because neither 1 nor $\{1,2\}$ is a suitable designated state for $\mathsf{f}(1,2)$. Making $\mathcal{A}$ quasi-deterministic produces the tree automaton

$\mathsf{a} \to 1 \mid 2 \mid 3 \mid 5$			
$\mathsf{f}(1,2) \to 1 \mid 3 \mid 4$	$\mathsf{f}(1,3) \to 1$	$\mathsf{f}(1,4) \to 1$	$\mathsf{f}(1,5) \to 4$
$\mathsf{f}(3,2) \to 1$	$\mathsf{f}(3,3) \to 1$	$\mathsf{f}(3,4) \to 1$	$\mathsf{f}(3,5) \to 1$
$\mathsf{f}(4,2) \to 4$	$\mathsf{f}(4,3) \to 1$	$\mathsf{f}(4,4) \to 1$	$\mathsf{f}(4,5) \to 4$
$\mathsf{f}(5,2) \to 4$	$\mathsf{f}(5,3) \to 1$	$\mathsf{f}(5,4) \to 1$	$\mathsf{f}(5,5) \to 4$

where $\{1,2\}$ is abbreviated by 3, $\{1,3\}$ is abbreviated by 4, and $\{1,2,3\}$ is abbreviated by 5. The final states are 1, 3, 4, and 5. The intermediate tree automaton constructed in step one of the quasi-determinisation procedure looks as follows:

$\mathsf{a} \to 5$			
$\mathsf{f}(\{1\},\{2\}) \to 4$	$\mathsf{f}(\{1\},\{3\}) \to \{1\}$	$\mathsf{f}(\{1\},4) \to \{1\}$	$\mathsf{f}(\{1\},5) \to 4$
$\mathsf{f}(\{3\},\{2\}) \to \{1\}$	$\mathsf{f}(\{3\},\{3\}) \to \{1\}$	$\mathsf{f}(\{3\},4) \to \{1\}$	$\mathsf{f}(\{3\},5) \to \{1\}$
$\mathsf{f}(4,\{2\}) \to 4$	$\mathsf{f}(4,\{3\}) \to \{1\}$	$\mathsf{f}(4,4) \to \{1\}$	$\mathsf{f}(4,5) \to 4$
$\mathsf{f}(5,\{2\}) \to 4$	$\mathsf{f}(5,\{3\}) \to \{1\}$	$\mathsf{f}(5,4) \to \{1\}$	$\mathsf{f}(5,5) \to 4$

During the construction of this tree automaton a fresh designated state 5 for the left-hand side a is introduced. It is easy to see that this step is unnecessary because adding a designated state for the left-hand side $\mathsf{f}(1,2)$ would suffice to make $\mathcal{A}$ quasi-deterministic.

3.5.2 Approximating Quasi-Deterministic Tree Automata

We now present a slightly modified version of the initial procedure which analysis the given tree automaton to ensure that as few transitions as possible are added during the quasi-determinisation process. To simplify the presentation we assume without loss of generality that all tree automata are *state-consistent*. A tree automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ has this property if there exists a subset $N \subset \mathbb{N}$ of natural numbers such that for all $q \in Q$ either $q \in N$ or $q \in 2^N$ with $|q| > 1$. So each state of the automaton corresponds either to some natural number or to some set of natural numbers. To ensure that newly added states do not violate this condition we use a function $\mathsf{flat}\colon 2^Q \to 2^N$, inductively defined as $\mathsf{flat}(\varnothing) = \varnothing$ and $\mathsf{flat}(\{p\} \cup P) = \mathsf{unif}_N(p) \cup \mathsf{flat}(P)$ for all $p \in Q$ and $P \subseteq 2^Q$, to modify states.

Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a state-consistent tree automaton and $N \subset \mathbb{N}$ the smallest finite set of natural numbers which guarantees that $\mathcal{A}$ is state-consistent. We transform $\mathcal{A}$ into a state-consistent and quasi-deterministic one without losing any transitions of Δ as follows:

1. Construct the deterministic tree automaton $\mathcal{A}' = (\mathcal{F}, Q', Q'_f, \Delta')$ defined as $Q' = 2^N$, $Q'_f = \{p \in Q' \mid p \supseteq \mathsf{unif}_N(q) \text{ for some } q \in Q_f\}$, and Δ' consisting of all transitions $f(q_1, \ldots, q_n) \to q$ with $q_1, \ldots, q_n \in Q'$ and $q = \mathsf{flat}(\{p \mid f(p_1, \ldots, p_n) \to p \in \Delta \text{ and } \mathsf{unif}_N(p_i) \subseteq q_i \text{ for all } i \in \{1, \ldots, n\}\})$.

2. Take the union of $\mathcal{A}$ and $\mathcal{A}'$ after identifying states $\{q\} \in Q'$ with $q \in Q$.

We continue the previous example.

Example 3.21. Let $\mathcal{A}$ be the tree automata of Example 3.20. Making $\mathcal{A}$ deterministic using the algorithm proposed in step one of the second procedure yields a tree automaton with final states $\{1\}$ and 3, and the transitions

$$\begin{aligned} &\mathsf{a} \to 3 \qquad && \mathsf{f}(\{1\},\{2\}) \to 3 \qquad && \mathsf{f}(\{1\},3) \to 3 \\ & && \mathsf{f}(3,\{2\}) \to 3 && \mathsf{f}(3,3) \to 3 \end{aligned}$$

where $\{1,2\}$ is abbreviated by 3. Combining those transitions with the ones contained in $\mathcal{A}$ after identifying $\{1\}$ with 1 and $\{2\}$ with 2 yields the quasi-deterministic tree automaton

$$\begin{aligned} &\mathsf{a} \to 1 \mid 2 \mid 3 \qquad && \mathsf{f}(1,2) \to 1 \mid 3 \qquad && \mathsf{f}(1,3) \to 1 \mid 3 \\ & && \mathsf{f}(3,2) \to 1 \mid 3 && \mathsf{f}(3,3) \to 1 \mid 3 \end{aligned}$$

where 3 is the designated state of all left-hand sides. The final states are 1 and 3. Compared to the quasi-deterministic tree automaton given in Example 3.20, only 3 instead of 14 transitions have been added to make $\mathcal{A}$ quasi-deterministic.

It remains to prove that the above procedure is correct. To this end we show that it transforms each state-consistent tree automaton $\mathcal{A}$ into a state-consistent and quasi-deterministic tree automaton that accepts at least all terms that are accepted by $\mathcal{A}$.

Theorem 3.22. *For every state-consistent tree automaton $\mathcal{A}$ there is a state-consistent and quasi-deterministic tree automaton $\mathcal{A}''$ such that $\mathcal{L}(\mathcal{A}'') \supseteq \mathcal{L}(\mathcal{A})$.*

Proof. Let $\mathcal{A}'' = (\mathcal{F}, Q'', Q''_f, \Delta'')$ be the tree automaton obtained from the tree automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ by applying the second quasi-determinism procedure. The inclusion $\mathcal{L}(\mathcal{A}'') \supseteq \mathcal{L}(\mathcal{A})$ holds trivially because $\Delta \subseteq \Delta''$ and $Q_f \subseteq Q''_f$. The proof that $\mathcal{A}''$ is quasi-deterministic is similar to the one of Theorem 3.19. $\square$

As already indicated by the previous theorem we do not necessarily have $\mathcal{L}(\mathcal{A}'') = \mathcal{L}(\mathcal{A})$ if we use the second procedure to transform a state-consistent tree automaton $\mathcal{A}$ into a state-consistent and quasi-deterministic tree automaton $\mathcal{A}''$. The reason is that the intermediate tree automaton $\mathcal{A}'$ constructed in step one of the procedure might accept more ground terms than $\mathcal{A}$. This behavior is illustrated in the next example.

Example 3.23. Let $\mathcal{A}$ be the state-consistent tree automaton consisting of the transitions

$$\mathsf{a} \to \{1,2\} \mid 3 \qquad \mathsf{b} \to 1 \mid \{2,3\} \qquad \mathsf{f}(\{1,2\},\{2,3\}) \to 4$$

and the final state 4. Making $\mathcal{A}$ deterministic using the approach described in step one of the second quasi-determinism procedure yields the tree automaton

$$\mathsf{a} \to \{1,2,3\} \qquad \mathsf{b} \to \{1,2,3\} \qquad \mathsf{f}(\{1,2,3\},\{1,2,3\}) \to \{4\}$$

with the final state $\{4\}$. It is easy to check that the term $\mathsf{f}(\mathsf{a},\mathsf{a})$ is accepted by the above tree automaton but not by $\mathcal{A}$.

Although the second procedure is inexact in the sense that it over-approximates the language of the tree automaton that should be made quasi-deterministic, it is in general a better choice than the first one. First of all, it changes only those parts of the given tree automaton which violate the quasi-compatibility requirement. So information regarding existing designated states are taken into consideration which ensures that much less new states and transitions have to be added in contrast to the first procedure. Secondly, the number of states added to the tree automaton to make it quasi-deterministic can be bounded by $2^{|N|}$ for some $N \subset \mathbb{N}$ even if the procedure is called several times. In case of tree automata completion, N consists of all states of the initial tree automaton as well as all states that have been added to solve compatibility violations. For the first procedure this need not be the case since each time the procedure is executed, an exponential increase of the number of states can happen. So in general we have at most $2^{|N|}$ states after the first call, $2^{2^{|N|}}$ states after the second call, etc. Of course, by using the second procedure it can always happen that the obtained language accepts some strings which have some inappropriate properties. However, in most cases the over-approximation of the language of the given tree automaton has no negative effects since the whole completion process is designed to over-approximate $\rightarrow^*_{\mathcal{R}}(L)$ for some TRS $\mathcal{R}$ and regular language L.

3.6 Summary

In this chapter we presented an alternative approach to cope with non-left-linear TRSs during tree automata completion. To this end we introduced *quasi-deterministic* tree automata because the common approach to handle non-left-linear rewrite rules by using deterministic tree automata turned out to be incompatible with tree automata completion. Last but not least we presented an effective *quasi-determinisation procedure* which ensures that the size of the final automaton is at most exponential in the number of states added during the completion process, including the states of the initial tree automaton.

Chapter 4

The Match-Bound Technique

The *match-bound technique*, introduced by Geser *et al.* in a sequence of papers [21, 22, 23, 24], is a relatively new and elegant approach to automatically prove the termination of rewrite systems. In order to obtain a termination certificate, it uses *automata techniques* to reason about the derivations induced by the given TRS $\mathcal{R}$. To this end, $\mathcal{R}$ is transformed into an enriched system, where function symbols are labeled with natural numbers. The key features of this new rewrite system are that every rewrite step increases the labels in the contracted redex and each original derivation can be simulated via the rewrite rules of the new system. If the labels occurring in rewriting sequences induced by the enriched system can be globally bounded, we can conclude that the original TRS $\mathcal{R}$ is terminating.

Initially, Geser, Hofbauer, and Waldmann introduced the match-bound technique for string rewriting [21]. For this particular class of TRSs, the used enrichment is deleting and hence regularity preserving [37]. Therefore, it is semi-decidable if there exists a global bound on the labels introduced by the enriched system [13]. Later on the method has been extended to left-linear TRSs by Geser, Hofbauer, Waldmann, and Zantema [24]. The key to this extension is the usage of *tree automata* to represent the terms that occur in derivations caused by the enriched system. Since for TRSs the underlying enrichment need not be regularity preserving, semi-decidability of the existence of a global bound is lost. So instead of constructing the exact set of terms that occur in derivations induced by the used enrichment, we use *tree automata completion* to approximate it [23]. The fact that the method has been implemented in several different termination provers [13, 26, 44, 59, 63] is a clear witness of the success of the approach.

The remainder of this chapter is organized as follows. After recalling some preliminary notions we formally introduce the match-bound technique in Section 4.2. In Section 4.3 we extend the method by removing the left-linearity restriction. This turns out to be surprisingly challenging because the theory on which the method is based does not work without further ado for non-left-linear rewrite systems. So-called *raise-rules* are introduced to solve this issue. After that, in Section 4.4 we show how *tree automata completion* can be used to automate the match-bound technique. To ensure that the approach works also for non-left-linear TRSs, the raise-rules need special care to enable the automata construction to terminate. Finally in Section 4.5 we increase the power of the match-bound technique by considering *right-hand sides of forward closures*.

The information presented in this chapter have already been published in the conference paper [40] and the journal paper [43]. However, compared to the results in [40, 43] the findings in Section 4.4 are based upon the optimized definition of quasi-deterministic tree automata introduced in Section 3.4.

4.1 Preliminaries

Let $\mathcal{F}$ be a finite signature. For a set $N \subseteq \mathbb{N}$ of natural numbers, the signature $\mathcal{F} \times N$ is abbreviated by $\mathcal{F}_N$. Here function symbols (f, n) with $f \in \mathcal{F}$ and $n \in N$ have the same arity as f and are written as f_n. To deal with function symbols contained in the signature $\mathcal{F}_N$ we use the mappings $\mathsf{lift}_c \colon \mathcal{F} \to \mathcal{F}_\mathbb{N}$, $\mathsf{base} \colon \mathcal{F}_\mathbb{N} \to \mathcal{F}$, and $\mathsf{height} \colon \mathcal{F}_\mathbb{N} \to \mathbb{N}$ defined as

$$\mathsf{lift}_c(f) = f_c \qquad \mathsf{base}(f_c) = f \qquad \mathsf{height}(f_c) = c$$

for all $f \in \mathcal{F}$ and $c \in \mathbb{N}$. The application of a function $\phi \in \{\mathsf{lift}_c, \mathsf{base}\}$ to a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ is defined as follows:

$$\phi(t) = \begin{cases} t & \text{if } t \text{ is a variable} \\ \phi(f)(\phi(t_1), \dots, \phi(t_n)) & \text{if } t = f(t_1, \dots, t_n) \end{cases}$$

These mappings are extended to sets of terms and TRSs in the obvious way. Let $\mathcal{M}\mathsf{ul}(\mathbb{N})$ denote the set of all finite multisets over $\mathbb{N}$. For any $M \in \mathcal{M}\mathsf{ul}(\mathbb{N})$ we write $M(n)$ to denote how often the number $n \in \mathbb{N}$ occurs in M. Let $M, N \in \mathcal{M}\mathsf{ul}(\mathbb{N})$ be two multisets. We write $M \cup N$ for the multiset sum of M and N where $(M \cup N)(n) = M(n) + N(n)$ for all $n \in \mathbb{N}$. The multiset difference $M \setminus N$ is defined as $(M \setminus N)(n) = M(n) - N(n)$ if $M(n) > N(n)$ and $(M \setminus N)(n) = 0$ otherwise, for all $n \in \mathbb{N}$. We write $M \succ_{\mathsf{mul}} N$ if there are multisets X and Y such that $N = (M \setminus X) \cup Y$, $X \neq \varnothing$, and for all $m \in Y$ there is a $n \in X$ such that $n < m$. We write $M \succeq_{\mathsf{mul}} N$ if $M \succ_{\mathsf{mul}} N$ or $M = N$. Let $\mathcal{F}$ be some signature. We extend the orderings $\succ_{\mathsf{mul}}$ and $\succeq_{\mathsf{mul}}$ to terms over the signature $\mathcal{F}_\mathbb{N}$ as follows: we have $s \succ_{\mathsf{mul}} t$ if $\mathcal{F}\mathsf{un}_\mathcal{M}(s) \succ_{\mathsf{mul}} \mathcal{F}\mathsf{un}_\mathcal{M}(t)$ and $s \succeq_{\mathsf{mul}} t$ if $\mathcal{F}\mathsf{un}_\mathcal{M}(s) \succeq_{\mathsf{mul}} \mathcal{F}\mathsf{un}_\mathcal{M}(t)$ for terms $s, t \in \mathcal{T}(\mathcal{F}_\mathbb{N}, \mathcal{V})$. Here $\mathcal{F}\mathsf{un}_\mathcal{M}(t)$ denotes the multiset of the heights of the function symbols that occur in the term t: $\mathcal{F}\mathsf{un}_\mathcal{M}(t) = \{\mathsf{height}(t(p)) \mid p \in \mathcal{P}\mathsf{os}_\mathcal{F}(t)\}$. Let $\mathcal{R}$ be a finite or infinite TRS over a finite or infinite signature $\mathcal{F}$. The *restriction* of $\mathcal{R}$ to some finite signature $\mathcal{G} \subseteq \mathcal{F}$ is defined as $\{l \to r \in \mathcal{R} \mid l, r \in \mathcal{T}(\mathcal{G}, \mathcal{V})\}$. We call $\mathcal{R}$ *locally terminating* if every restriction of $\mathcal{R}$ to some finite signature $\mathcal{G} \subseteq \mathcal{F}$ is terminating.

Example 4.1. Consider the infinite TRS $\mathcal{R} = \{\mathsf{f}_c(x) \to \mathsf{f}_{c+1}(x) \mid c \geqslant 0\}$ over the signature $\mathcal{F} = \{\mathsf{f}_c \mid c \geqslant 0\}$. It is easy to see that $\mathcal{R}$ is both non-terminating and locally terminating.

4.2 Bounds for Left-Linear TRSs

To prove the termination of a TRS $\mathcal{R}$ over a signature $\mathcal{F}$ using the match-bound technique [21, 24], first an enriched system over the new signature $\mathcal{F}_\mathbb{N}$ is constructed that simulates the original derivations. The idea behind the new TRS is that after a rewrite step, the minimal height of the rewritten part is greater than the minimal height of the contracted redex. Below we introduce three different enrichments that have been proposed in the literature [24].

Let t be a term in $\mathcal{T}(\mathcal{F},\mathcal{V})$ and $V \subseteq \mathcal{V}\mathsf{ar}(t)$ a set of variables. A position $p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(t)$ is a *roof position* in t for V if $V \subseteq \mathcal{V}\mathsf{ar}(t|_p)$. The set of all roof positions in t for V is denoted by $\mathcal{P}\mathsf{os}_{\mathcal{R}}(t,V)$. Let l and r be two terms in $\mathcal{T}(\mathcal{F},\mathcal{V})$. The mappings top, roof, and match are defined as follows:

$$\mathsf{top}(l,r) = \{\epsilon\} \qquad \mathsf{roof}(l,r) = \mathcal{P}\mathsf{os}_{\mathcal{R}}(l,\mathcal{V}\mathsf{ar}(r)) \qquad \mathsf{match}(l,r) = \mathcal{P}\mathsf{os}_{\mathcal{F}}(l)$$

Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$ and e a function that maps every rewrite rule $l \to r \in \mathcal{R}$ to a nonempty subset of $\mathcal{P}\mathsf{os}_{\mathcal{F}}(l)$. The TRS $e(\mathcal{R})$ over the signature $\mathcal{F}_{\mathbb{N}}$ consists of all rewrite rules $l' \to \mathsf{lift}_c(r)$ for which there exists a rule $l \to r \in \mathcal{R}$ such that $\mathsf{base}(l') = l$ and $c = 1+\min\{\mathsf{height}(l'(p)) \mid p \in e(l,r)\}$. Let $c \in \mathbb{N}$. The restriction of $e(\mathcal{R})$ to the signature $\mathcal{F}_{\{0,\ldots,c\}}$ is denoted by $e_c(\mathcal{R})$. Furthermore, we write $e(l \to r)$ for $e(\{l \to r\})$.

Example 4.2. Let $\mathcal{R}$ be the TRS consisting of the rewrite rule $\mathsf{f}(\mathsf{g}(x,\mathsf{h}(y))) \to \mathsf{g}(\mathsf{h}(\mathsf{f}(x)),y)$. Then $\mathsf{top}(\mathcal{R})$ contains the rewrite rules

$$\begin{array}{ll}
\mathsf{f}_0(\mathsf{g}_0(x,\mathsf{h}_0(y))) \to \mathsf{g}_1(\mathsf{h}_1(\mathsf{f}_1(x)),y) & \mathsf{f}_0(\mathsf{g}_0(x,\mathsf{h}_1(y))) \to \mathsf{g}_1(\mathsf{h}_1(\mathsf{f}_1(x)),y)\\
\mathsf{f}_0(\mathsf{g}_1(x,\mathsf{h}_0(y))) \to \mathsf{g}_1(\mathsf{h}_1(\mathsf{f}_1(x)),y) & \mathsf{f}_1(\mathsf{g}_0(x,\mathsf{h}_0(y))) \to \mathsf{g}_2(\mathsf{h}_2(\mathsf{f}_2(x)),y)\\
\mathsf{f}_1(\mathsf{g}_1(x,\mathsf{h}_0(y))) \to \mathsf{g}_2(\mathsf{h}_2(\mathsf{f}_2(x)),y) & \cdots
\end{array}$$

$\mathsf{roof}(\mathcal{R})$ contains

$$\begin{array}{ll}
\mathsf{f}_0(\mathsf{g}_0(x,\mathsf{h}_0(y))) \to \mathsf{g}_1(\mathsf{h}_1(\mathsf{f}_1(x)),y) & \mathsf{f}_0(\mathsf{g}_0(x,\mathsf{h}_1(y))) \to \mathsf{g}_1(\mathsf{h}_1(\mathsf{f}_1(x)),y)\\
\mathsf{f}_0(\mathsf{g}_1(x,\mathsf{h}_0(y))) \to \mathsf{g}_1(\mathsf{h}_1(\mathsf{f}_1(x)),y) & \mathsf{f}_1(\mathsf{g}_0(x,\mathsf{h}_0(y))) \to \mathsf{g}_1(\mathsf{h}_1(\mathsf{f}_1(x)),y)\\
\mathsf{f}_1(\mathsf{g}_1(x,\mathsf{h}_0(y))) \to \mathsf{g}_2(\mathsf{h}_2(\mathsf{f}_2(x)),y) & \cdots
\end{array}$$

and the rewrite rules

$$\begin{array}{ll}
\mathsf{f}_0(\mathsf{g}_0(x,\mathsf{h}_0(y))) \to \mathsf{g}_1(\mathsf{h}_1(\mathsf{f}_1(x)),y) & \mathsf{f}_0(\mathsf{g}_0(x,\mathsf{h}_1(y))) \to \mathsf{g}_1(\mathsf{h}_1(\mathsf{f}_1(x)),y)\\
\mathsf{f}_0(\mathsf{g}_1(x,\mathsf{h}_0(y))) \to \mathsf{g}_1(\mathsf{h}_1(\mathsf{f}_1(x)),y) & \mathsf{f}_1(\mathsf{g}_0(x,\mathsf{h}_0(y))) \to \mathsf{g}_1(\mathsf{h}_1(\mathsf{f}_1(x)),y)\\
\mathsf{f}_1(\mathsf{g}_1(x,\mathsf{h}_0(y))) \to \mathsf{g}_1(\mathsf{h}_1(\mathsf{f}_1(x)),y) & \cdots
\end{array}$$

belong to the TRS $\mathsf{match}(\mathcal{R})$. Note that all three TRSs have infinitely many rewrite rules.

To be able to use $e(\mathcal{R})$ for proving the termination of the TRS $\mathcal{R}$ it must be guaranteed that $\mathcal{R}$ is terminating whenever $e(\mathcal{R})$ is terminating. In [24] the following result has been proved.

Lemma 4.3. *Let $\mathcal{R}$ be a TRS. The TRSs $\mathsf{top}(\mathcal{R})$ and $\mathsf{roof}(\mathcal{R})$ are locally terminating. If $\mathcal{R}$ is right-linear then the TRS $\mathsf{match}(\mathcal{R})$ is locally terminating.* ☐

By definition, the TRS $e(\mathcal{R})$ has an infinite signature and consists of infinitely many rewrite rules whenever $\mathcal{R} \neq \varnothing$. The idea is now to check whether there exists a finite subset of $e(\mathcal{R})$ which simulates all derivations of the original TRS $\mathcal{R}$. Let $e \in \{\mathsf{top}, \mathsf{roof}, \mathsf{match}\}$ and L a language. A TRS $\mathcal{R}$ is called *e-bounded* for L if there exists a $c \in \mathbb{N}$ such that the maximum height of function symbols occurring in terms in $\rightarrow^*_{e(\mathcal{R})}(\mathsf{lift}_0(L))$ is at most c. If we want to indicate the bound c, we say that $\mathcal{R}$ is e-bounded for L *by* c. In the following we do not mention L if we have the set of all ground terms in mind. The main result from [24], underlying the match-bound technique, is stated below.

Theorem 4.4. *Let $\mathcal{R}$ be a left-linear TRS and L some language. If $\mathcal{R}$ is top-bounded, roof-bounded, or both right-linear and match-bounded for L then $\mathcal{R}$ is terminating on L.* $\square$

In [24] it is shown that match-bounds are strictly more powerful than roof-bounds and roof-bounds are strictly more powerful than top-bounds. So in general one would prefer $\mathsf{roof}(\mathcal{R})$ to $\mathsf{top}(\mathcal{R})$, and one will use $\mathsf{match}(\mathcal{R})$ for non-duplicating TRSs. The reason for introducing $\mathsf{top}(\mathcal{R})$ is that we have to resort to it in Section 5.2. We conclude this section with two examples to illustrate this hierarchy.

Example 4.5. Consider the TRS $\mathcal{R}$ over the signature $\mathcal{F} = \{\mathsf{a}, \mathsf{f}, \mathsf{g}, \mathsf{h}\}$ of Example 4.2. We show that $\mathcal{R}$ is not top-bounded for $\mathcal{T}(\mathcal{F})$. Consider the substitutions $\sigma = \{x \mapsto \mathsf{g}_0(x, \mathsf{h}_0(\mathsf{a}_0))\}$, $\tau = \{x \mapsto \mathsf{a}_0\}$, $\mu_i = \{x \mapsto \mathsf{g}_i(\mathsf{h}_i(x), \mathsf{a}_0)\}$, and $\nu_i = \{x \mapsto \mathsf{f}_i(\mathsf{a}_0)\}$ for all $i \geqslant 1$. We have

$$\mathsf{f}_0(\mathsf{g}_0(x, \mathsf{h}_0(\mathsf{a}_0)))\sigma^{i-1}\tau \rightarrow^*_{\mathsf{top}(\mathcal{R})} \mathsf{g}_1(\mathsf{h}_1(x), \mathsf{a}_0)\mu_2 \cdots \mu_i \nu_i$$

for all $i \geqslant 1$. However, $\mathcal{R}$ is roof-bounded by 2 and match-bounded by 1. In Subsection 4.4.1 it is explained how this can be automatically checked.

Example 4.6. The TRS $\mathcal{R}$ consisting of the rewrite rule $\mathsf{f}(x, \mathsf{g}(y)) \rightarrow \mathsf{f}(x, y)$ over the signature $\mathcal{F} = \{\mathsf{a}, \mathsf{f}, \mathsf{g}\}$ is match-bounded for $\mathcal{T}(\mathcal{F})$ by 1 but neither top-bounded nor roof-bounded. The latter result follows from the fact that $\mathsf{f}_0(\mathsf{a}_0, \mathsf{g}_0^n(\mathsf{a}_0)) \rightarrow^*_{\mathcal{S}} \mathsf{f}_n(\mathsf{a}_0, \mathsf{a}_0)$ where $\mathcal{S} = \mathsf{top}(\mathcal{R}) = \mathsf{roof}(\mathcal{R})$.

4.3 Raise-Bounds for Non-Left-Linear TRSs

The first problem that arises if one wants to extend the match-bound technique to arbitrary TRSs is that e-bounded TRSs need not be terminating in the presence of non-left-linear rewrite rules.

Example 4.7. Consider the non-terminating TRS $\mathcal{R} = \{\mathsf{f}(x, x) \rightarrow \mathsf{f}(\mathsf{a}, x)\}$. The TRSs $\mathsf{match}(\mathcal{R})$, $\mathsf{roof}(\mathcal{R})$, and $\mathsf{top}(\mathcal{R})$ coincide and consist of the rules

$$\mathsf{f}_i(x, x) \rightarrow \mathsf{f}_{i+1}(\mathsf{a}_{i+1}, x)$$

for all $i \geqslant 0$. It is not difficult to see that with these rewrite rules we can never reach height 2 starting from a term in $\mathcal{T}(\{\mathsf{a}_0, \mathsf{f}_0\})$. For instance, we have $\mathsf{f}(\mathsf{a}, \mathsf{a}) \rightarrow_{\mathcal{R}} \mathsf{f}(\mathsf{a}, \mathsf{a}) \rightarrow_{\mathcal{R}} \mathsf{f}(\mathsf{a}, \mathsf{a})$ but after the step $\mathsf{f}_0(\mathsf{a}_0, \mathsf{a}_0) \rightarrow_{e(\mathcal{R})} \mathsf{f}_1(\mathsf{a}_1, \mathsf{a}_0)$ we are stuck because $\mathsf{a}_0 \neq \mathsf{a}_1$. Hence $\mathcal{R}$ is e-bounded by 1 for all $e \in \{\mathsf{top}, \mathsf{roof}, \mathsf{match}\}$.

The problem is that even though every single $\rightarrow_{\mathcal{R}}$-step can be simulated by an $\rightarrow_{e(\mathcal{R})}$-step, this does not hold for consecutive $\rightarrow_{\mathcal{R}}$-steps. To overcome this problem we introduce *raise-rules* which increase the heights of function symbols.

Definition 4.8. Let $\mathcal{F}$ be a signature. The TRS $\mathsf{raise}(\mathcal{F})$ over the signature $\mathcal{F}_{\mathbb{N}}$ consists of all rules

$$f_c(x_1, \ldots, x_n) \rightarrow f_{c+1}(x_1, \ldots, x_n)$$

with f an n-ary function symbol in $\mathcal{F}$, $c \in \mathbb{N}$, and $x_1, \ldots, x_n$ pairwise distinct variables. The restriction of $\mathsf{raise}(\mathcal{F})$ to the signature $\mathcal{F}_{\{0,\ldots,c\}}$ is denoted by $\mathsf{raise}_c(\mathcal{F})$. For terms $s, t \in \mathcal{T}(\mathcal{F}_{\mathbb{N}}, \mathcal{V})$ we write $s \succeq t$ if $t \rightarrow^*_{\mathsf{raise}(\mathcal{F})} s$ and $s \uparrow t$ for the least term u with $u \succeq s$ and $u \succeq t$. The latter notion is extended to $\uparrow S$ for finite nonempty sets $S \subset \mathcal{T}(\mathcal{F}_{\mathbb{N}}, \mathcal{V})$ in the obvious way. Note that $\uparrow S$ is undefined whenever S contains two terms s and t such that $\mathsf{base}(s) \neq \mathsf{base}(t)$.

The following result corresponds to Lemma 4.3. The right-linearity condition of the TRS $\mathsf{match}(\mathcal{R})$ is weakened to non-duplication in order to cover more non-left-linear TRSs.

Lemma 4.9. *Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$. The TRSs $\mathsf{top}(\mathcal{R}) \cup \mathsf{raise}(\mathcal{F})$ and $\mathsf{roof}(\mathcal{R}) \cup \mathsf{raise}(\mathcal{F})$ are locally terminating. If $\mathcal{R}$ is non-duplicating then the TRS $\mathsf{match}(\mathcal{R}) \cup \mathsf{raise}(\mathcal{F})$ is locally terminating.*

Proof. First we consider $e(\mathcal{R}) \cup \mathsf{raise}(\mathcal{F})$ with $e \in \{\mathsf{top}, \mathsf{roof}\}$. From the proof of [24, Lemma 16] we know that the rewrite rules in $e(\mathcal{R})$ are oriented from left to right by the recursive path order [8] induced by the precedence $\succ$ on $\mathcal{F}_{\mathbb{N}}$ defined as $f \succ g$ if and only if $\mathsf{height}(f) < \mathsf{height}(g)$. The same holds for the rules in $\mathsf{raise}(\mathcal{F})$. Since the precedence $\succ$ is well-founded on any finite subset of $\mathcal{F}_{\mathbb{N}}$, we conclude that $e(\mathcal{R}) \cup \mathsf{raise}(\mathcal{F})$ is locally terminating. Next we show that $\mathsf{match}(\mathcal{R}) \cup \mathsf{raise}(\mathcal{F})$ is locally terminating. From the proof of [24, Lemma 17] we know that for a non-duplicating TRS $\mathcal{R}$, $s \succ_{\mathsf{mul}} t$ whenever $s \rightarrow_{\mathsf{match}(\mathcal{R})} t$ for terms $s, t \in \mathcal{T}(\mathcal{F}_{\mathbb{N}}, \mathcal{V})$. If $s \rightarrow_{\mathsf{raise}(\mathcal{F})} t$ then $\mathcal{F}\mathsf{un}_{\mathcal{M}}(t) = (\mathcal{F}\mathsf{un}_{\mathcal{M}}(s) \setminus \{c\}) \cup \{c+1\}$ for some height $c \in \mathbb{N}$, and thus $\mathcal{F}\mathsf{un}_{\mathcal{M}}(s) \succ_{\mathsf{mul}} \mathcal{F}\mathsf{un}_{\mathcal{M}}(t)$. Since $\succ_{\mathsf{mul}}$ inherits well-foundedness from $<$, we conclude that the TRS $\mathsf{match}(\mathcal{R}) \cup \mathsf{raise}(\mathcal{F})$ is locally terminating. Note that $<$ is well-founded on $\{0, \ldots, c\}$. □

In order to use $e(\mathcal{R}) \cup \mathsf{raise}(\mathcal{F})$ to infer termination of the TRS $\mathcal{R}$, we have to restrict the rules of $\mathsf{raise}(\mathcal{F})$ to those that are really needed to simulate derivations in $\mathcal{R}$ because $\mathsf{raise}(\mathcal{F})$ is non-terminating. We do this by defining a new relation $\xrightarrow{\mathsf{r}}_{e(\mathcal{R})}$ in which the necessary raise-steps are built in. The idea is that $s \xrightarrow{\mathsf{r}}_{e(\mathcal{R})} t$ if t can be obtained from s by doing the minimum number of raise-steps to ensure the applicability of a non-left-linear rewrite rule in $e(\mathcal{R})$.

Definition 4.10. Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$. We define the relation $\xrightarrow{\mathsf{r}}_{e(\mathcal{R})}$ on $\mathcal{T}(\mathcal{F}_{\mathbb{N}}, \mathcal{V})$ as follows: $s \xrightarrow{\mathsf{r}}_{e(\mathcal{R})} t$ if and only if there exist a rewrite rule $l \rightarrow r \in e(\mathcal{R})$, a position $p \in \mathcal{P}\mathsf{os}(s)$, a context C, and terms $s_1, \ldots, s_n$ such that $l = C[x_1, \ldots, x_n]$ with all variables displayed, $s|_p = C[s_1, \ldots, s_n]$,

$\mathsf{base}(s_i) = \mathsf{base}(s_j)$ whenever $x_i = x_j$ for all $i, j \in \{1, \ldots, n\}$, and $t = s[r\sigma]_p$. Here the substitution σ is defined as follows:

$$\sigma(x) = \begin{cases} \uparrow\{s_i \mid x_i = x \text{ with } i \in \{1, \ldots, n\}\} & \text{if } x \in \{x_1, \ldots, x_n\} \\ x & \text{otherwise} \end{cases}$$

Note that $\xrightarrow{\mathsf{r}}_{e(\mathcal{R})} = \rightarrow_{e(\mathcal{R})}$ for left-linear TRSs $\mathcal{R}$. The following example illustrates how implicit raise-steps are used in $\xrightarrow{\mathsf{r}}_{e(\mathcal{R})}$ to simulate original derivations.

Example 4.11. Consider the TRS $\mathcal{R}$ consisting of the rewrite rules $\mathsf{g}(x, x) \rightarrow \mathsf{b}$ and $\mathsf{f}(x, x) \rightarrow \mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{a}, x))$. With the rewrite rules

$$\mathsf{f}_0(x, x) \rightarrow \mathsf{f}_1(\mathsf{a}_1, \mathsf{g}_1(\mathsf{a}_1, x)) \qquad \mathsf{g}_0(x, x) \rightarrow \mathsf{b}_1 \qquad \mathsf{g}_1(x, x) \rightarrow \mathsf{b}_2$$

of the TRS $\mathsf{match}(\mathcal{R})$, arbitrary derivations in $\mathcal{R}$ can be simulated using the rewrite relation $\xrightarrow{\mathsf{r}}_{\mathsf{match}(\mathcal{R})}$. For instance,

$$\begin{aligned} \mathsf{f}(\mathsf{f}(\mathsf{a}, \mathsf{a}), \mathsf{f}(\mathsf{a}, \mathsf{b})) &\rightarrow_{\mathcal{R}} \mathsf{f}(\mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{a}, \mathsf{a})), \mathsf{f}(\mathsf{a}, \mathsf{b})) \\ &\rightarrow_{\mathcal{R}} \mathsf{f}(\mathsf{f}(\mathsf{a}, \mathsf{b}), \mathsf{f}(\mathsf{a}, \mathsf{b})) \\ &\rightarrow_{\mathcal{R}} \mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{a}, \mathsf{f}(\mathsf{a}, \mathsf{b}))) \end{aligned}$$

is turned into the following rewrite sequence:

$$\begin{aligned} \mathsf{f}_0(\mathsf{f}_0(\mathsf{a}_0, \mathsf{a}_0), \mathsf{f}_0(\mathsf{a}_0, \mathsf{b}_0)) &\xrightarrow{\mathsf{r}}_{\mathsf{match}(\mathcal{R})} \mathsf{f}_0(\mathsf{f}_1(\mathsf{a}_1, \mathsf{g}_1(\mathsf{a}_1, \mathsf{a}_0)), \mathsf{f}_0(\mathsf{a}_0, \mathsf{b}_0)) \\ &\xrightarrow{\mathsf{r}}_{\mathsf{match}(\mathcal{R})} \mathsf{f}_0(\mathsf{f}_1(\mathsf{a}_1, \mathsf{b}_2), \mathsf{f}_0(\mathsf{a}_0, \mathsf{b}_0)) \\ &\xrightarrow{\mathsf{r}}_{\mathsf{match}(\mathcal{R})} \mathsf{f}_1(\mathsf{a}_1, \mathsf{g}_1(\mathsf{a}_1, \mathsf{f}_1(\mathsf{a}_1, \mathsf{b}_2))) \end{aligned}$$

Here the raise-rules

$$\mathsf{a}_0 \rightarrow \mathsf{a}_1 \qquad \mathsf{b}_0 \rightarrow \mathsf{b}_1 \qquad \mathsf{b}_1 \rightarrow \mathsf{b}_2 \qquad \mathsf{f}_0(x, y) \rightarrow \mathsf{f}_1(x, y)$$

are used implicitly to enable the application of the non-left-linear rules in $\mathsf{match}(\mathcal{R})$.

Definition 4.12. A TRS $\mathcal{R}$ is called *e-raise-bounded* for a language L if there exists a $c \in \mathbb{N}$ such that the maximum height of function symbols occurring in terms belonging to $\xrightarrow{\mathsf{r}}^*_{e(\mathcal{R})}(\mathsf{lift}_0(L))$ is at most c.

Note that e-raise-boundedness coincides with e-boundedness for left-linear TRSs. The quintessence of the next lemma is that every derivation caused by the TRS $\mathcal{R}$ can be simulated using the rewrite relation $\xrightarrow{\mathsf{r}}_{e(\mathcal{R})}$. This result is used to infer termination from e-raise-boundedness in Theorem 4.15.

Lemma 4.13. *Let $\mathcal{R}$ be a TRS. If $s \rightarrow_{\mathcal{R}} t$ then for all terms s' with $\mathsf{base}(s') = s$ there exists a term t' such that $\mathsf{base}(t') = t$ and $s' \xrightarrow{\mathsf{r}}_{e(\mathcal{R})} t'$.*

Proof. Straightforward. □

An immediate consequence of the above lemma is that by using the rewrite relation $\xrightarrow{\mathsf{r}}$ instead of $\rightarrow$, heights can no longer be responsible for the inapplicability of non-left-linear rewrite rules.

Example 4.14. Consider the TRS $\mathcal{R}$ over the signature $\mathcal{F} = \{\mathsf{a}, \mathsf{f}\}$ of Example 4.7. Using the rewrite relation $\xrightarrow{\mathsf{r}}_{e(\mathcal{R})}$ instead of $\rightarrow_{e(\mathcal{R})}$ we obtain the infinite rewrite sequence

$$\mathsf{f}_0(\mathsf{a}_0, \mathsf{a}_0) \xrightarrow{\mathsf{r}}_{e(\mathcal{R})} \mathsf{f}_1(\mathsf{a}_1, \mathsf{a}_0) \xrightarrow{\mathsf{r}}_{e(\mathcal{R})} \mathsf{f}_2(\mathsf{a}_2, \mathsf{a}_1) \xrightarrow{\mathsf{r}}_{e(\mathcal{R})} \mathsf{f}_3(\mathsf{a}_3, \mathsf{a}_2) \xrightarrow{\mathsf{r}}_{e(\mathcal{R})} \cdots$$

for all $e \in \{\mathsf{top}, \mathsf{roof}, \mathsf{match}\}$. Hence $\mathcal{R}$ is not e-raise-bounded for any $L \subseteq \mathcal{T}(\mathcal{F})$ that contains $\mathsf{f}(\mathsf{a}, \mathsf{a})$.

Using Lemma 4.13 we are now ready to show that a TRS is terminating whenever it is e-raise-bounded. The following result extends Theorem 4.4.

Theorem 4.15. *Let $\mathcal{R}$ be a TRS and L a language. If $\mathcal{R}$ is top-raise-bounded or roof-raise-bounded for L then $\mathcal{R}$ is terminating on L. If $\mathcal{R}$ is non-duplicating and match-raise-bounded for L then $\mathcal{R}$ is terminating on L.*

Proof. Assume to the contrary that there exists an infinite rewrite sequence

$$t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} t_3 \rightarrow_{\mathcal{R}} t_4 \rightarrow_{\mathcal{R}} \cdots$$

with $t_1 \in L$. With help of Lemma 4.13 this sequence is lifted to an infinite $\xrightarrow{\mathsf{r}}_{e(\mathcal{R})}$ rewrite sequence starting from $\mathsf{lift}_0(t_1)$. Since $\mathcal{R}$ is e-raise-bounded for L, all terms in this latter sequence belong to $\mathcal{T}(\mathcal{F}_{\{0,\dots,c\}})$ for some $c \in \mathbb{N}$. Hence the employed rules must come from $e_c(\mathcal{R}) \cup \mathsf{raise}_c(\mathcal{F})$ and therefore $e_c(\mathcal{R}) \cup \mathsf{raise}_c(\mathcal{F})$ is non-terminating. However, this is impossible because $e(\mathcal{R}) \cup \mathsf{raise}(\mathcal{F})$ is locally terminating according to Lemma 4.9. □

We conclude this section with an example.

Example 4.16. The TRS $\mathcal{R}$ over the signature $\mathcal{F} = \{\mathsf{a}, \mathsf{b}, \mathsf{f}, \mathsf{g}\}$ of Example 4.11 can be shown to be match-raise-bounded for $\mathcal{T}(\mathcal{F})$ by 2 and hence terminating by Theorem 4.15. In order to verify this bound, one has to cope with the TRS $\mathsf{match}(\mathcal{R})$ on the one hand and with the rewrite relation $\xrightarrow{\mathsf{r}}$ on the other hand. How this can be done automatically is explained in Subsections 4.4.1 and 4.4.2.

4.4 Automation

In the following part we shortly recapitulate how we can use tree automata completion to prove that a given TRS is e(-raise)-bounded. At first we present the classical approach using compatible tree automata. After that we introduce an optimized version of compatible tree automata—so called *quasi-compatible* tree automata—which take the heights of the function symbols into account to achieve smaller tree automata.

4.4.1 Compatible Tree Automata

In order to prove automatically that a left-linear TRS is e-bounded for some language L we use compatible tree automata [24] as presented in Chapter 3. That is, given some left-linear TRS $\mathcal{R}$ we try to construct a tree automaton $\mathcal{A}$ that is compatible with $e(\mathcal{R})$ and $\mathsf{lift}_0(L)$ by solving all violations of the compatibility requirement. The following result originates from [24].

Theorem 4.17. *Let $\mathcal{R}$ be a left-linear TRS and L a language. If there is a tree automaton $\mathcal{A}$ which is compatible with $e(\mathcal{R})$ and $\mathsf{lift}_0(L)$ then $\mathcal{R}$ is e-bounded for L.* $\square$

The use of tree automata completion to automate the match-bound technique causes a severe problem. If $\mathcal{R}$ is not e-bounded for L, $e(\mathcal{R})$ consists of infinitely many rewrite rules over an infinite signature. So the completion procedure will not terminate and hence we will never obtain a compatible tree automaton that approximates $\to^*_{e(\mathcal{R})}(\mathsf{lift}_0(L))$. To overcome this problem we just limit the time that is spent to construct a compatible tree automaton. As soon as the time is up, we conclude that the e-boundedness of $\mathcal{R}$ for L cannot be proved.

Example 4.18. Let $\mathcal{R}$ be the TRS over the signature $\mathcal{F} = \{\mathsf{a}, \mathsf{f}, \mathsf{g}, \mathsf{h}\}$ of Example 4.2. We show that $\mathcal{R}$ is match-bounded for $\mathcal{T}(\mathcal{F})$ by 1. As starting point we use the tree automaton consisting of the final state 1 and the transitions

$$\mathsf{a}_0 \to 1 \qquad \mathsf{f}_0(1) \to 1 \qquad \mathsf{g}_0(1,1) \to 1 \qquad \mathsf{h}_0(1) \to 1$$

accepting the language $\mathsf{lift}_0(\mathcal{T}(\mathcal{F}))$. The first compatibility violation that we solve is caused by the rewrite rule $\mathsf{f}_0(\mathsf{g}_0(x, \mathsf{h}_0(y))) \to_{\mathsf{match}(\mathcal{R})} \mathsf{g}_1(\mathsf{h}_1(\mathsf{f}_1(x)), y)$. We have $\mathsf{f}_0(\mathsf{g}_0(1, \mathsf{h}_0(1))) \to^* 1$ but not $\mathsf{g}_1(\mathsf{h}_1(\mathsf{f}_1(1)), 1) \to^* 1$. In order to solve this compatibility violation we add the fresh states 2 and 3 as well as the transitions $\mathsf{f}_1(1) \to 2$, $\mathsf{h}_1(2) \to 3$, and $\mathsf{g}_1(3,1) \to 1$. This gives rise to a new compatibility violation: $\mathsf{f}_1(\mathsf{g}_1(x, \mathsf{h}_0(y))) \to_{\mathsf{match}(\mathcal{R})} \mathsf{g}_1(\mathsf{h}_1(\mathsf{f}_1(x)), y)$ and $\mathsf{f}_1(\mathsf{g}_1(3, \mathsf{h}_0(1))) \to^* 2$ but not $\mathsf{g}_1(\mathsf{h}_1(\mathsf{f}_1(3)), 1) \to^* 2$. To establish the missing path we reuse the transition $\mathsf{h}_1(2) \to 3$ and just add the transitions $\mathsf{f}_1(3) \to 2$ and $\mathsf{g}_1(3,1) \to 2$. After that step the constructed tree automaton is compatible with $\mathsf{match}(\mathcal{R})$ and $\mathsf{lift}_0(\mathcal{T}(\mathcal{F}))$. Hence $\mathcal{R}$ is match-bounded for $\mathcal{T}(\mathcal{F})$ by 1 and therefore $\mathcal{R}$ is terminating.

To use Theorem 4.15 for proving termination it is necessary to construct a language which consists of at least all terms that are reachable from $\mathsf{lift}_0(L)$ via $\xrightarrow{\mathsf{r}}_{e(\mathcal{R})}$. As before we do that by using compatible tree automata. To cope with non-left-linear TRSs we use quasi-deterministic tree automata. Remember that the reason why we prefer quasi-deterministic tree automata over deterministic automata is the importance of preserving existing transitions during the construction of a compatible tree automaton, as explained in Section 3.4.

Example 4.19. Consider the TRS $\mathcal{R}$ over the signature $\mathcal{F} = \{\mathsf{a}, \mathsf{b}, \mathsf{f}\}$ consisting of the rewrite rules

$$\mathsf{f}(x,x) \to \mathsf{f}(\mathsf{a},\mathsf{b}) \qquad \mathsf{f}(\mathsf{a},\mathsf{a}) \to \mathsf{a} \qquad \mathsf{f}(\mathsf{b},\mathsf{b}) \to \mathsf{b}$$

as well as the initial tree automaton with the transitions

$$\mathsf{a}_0 \to 1 \qquad \mathsf{b}_0 \to 1 \qquad \mathsf{f}_0(1,1) \to 1$$

and the final state 1, accepting the language $\mathsf{lift}_0(\mathcal{T}(\mathcal{F}))$. We construct a quasi-deterministic tree automaton that is compatible with the TRS $\mathsf{match}(\mathcal{R})$ and the language $\mathsf{lift}_0(\mathcal{T}(\mathcal{F}))$. Since $\mathsf{f}_0(\mathsf{a}_0, \mathsf{a}_0) \to_{\mathsf{match}(\mathcal{R})} \mathsf{a}_1$ and $\mathsf{f}_0(\mathsf{a}_0, \mathsf{a}_0) \to^* 1$, we add the transition $\mathsf{a}_1 \to 1$. Similarly, $\mathsf{f}_0(\mathsf{b}_0, \mathsf{b}_0) \to_{\mathsf{match}(\mathcal{R})} \mathsf{b}_1$ gives rise to the transition $\mathsf{b}_1 \to 1$. Next we consider $\mathsf{f}_0(x, x) \to_{\mathsf{match}(\mathcal{R})} \mathsf{f}_1(\mathsf{a}_1, \mathsf{b}_1)$ with $\mathsf{f}_0(1, 1) \to 1$. In order to ensure $\mathsf{f}_1(\mathsf{a}_1, \mathsf{b}_1) \to^* 1$ we add the new states 2 and 3 and the transitions $\mathsf{a}_1 \to 2$, $\mathsf{b}_1 \to 3$, and $\mathsf{f}_1(2, 3) \to 1$. Making these transitions quasi-deterministic produces an automaton consisting of the final states 1, 4, and 5 as well as the transitions

$$\begin{array}{rrrr}
\mathsf{a}_0 \to 1 & \mathsf{b}_0 \to 1 & & \mathsf{f}_0(1,1) \to 1 \\
\mathsf{a}_1 \to 1 \mid 2 \mid 4 & \mathsf{b}_1 \to 1 \mid 3 \mid 5 & & \mathsf{f}_1(2,3) \to 1 \\
\mathsf{f}_0(1,4) \to 1 & \mathsf{f}_0(1,5) \to 1 & \mathsf{f}_0(4,1) \to 1 & \mathsf{f}_0(5,1) \to 1 \\
\mathsf{f}_0(4,4) \to 1 & \mathsf{f}_0(4,5) \to 1 & \mathsf{f}_0(5,4) \to 1 & \mathsf{f}_0(5,5) \to 1 \\
\mathsf{f}_1(2,5) \to 1 & \mathsf{f}_1(4,3) \to 1 & \mathsf{f}_1(4,5) \to 1 &
\end{array}$$

which is compatible with $\mathsf{match}(\mathcal{R})$ and $\mathsf{lift}_0(\mathcal{T}(\mathcal{F}))$. Here 4 (abbreviating $\{1, 2\}$) is the designated state for a_1 and 5 (abbreviating $\{1, 3\}$) is the designated state for b_1. The transitions in the last three rows are added to satisfy the condition of Definition 3.10. If we would try to construct a deterministic tree automaton that is compatible with $\mathsf{match}(\mathcal{R})$ and $\mathsf{lift}_0(\mathcal{T}(\mathcal{F}))$ we would never succeed. To see this assume that we have already solved the first two compatibility violations. To solve the third one, we may reuse one or both of the transitions $\mathsf{a}_1 \to 1$ and $\mathsf{b}_1 \to 1$. Let us consider the various alternatives.

- If we reuse both transitions then we only need to add the transition $\mathsf{f}_1(1, 1) \to 1$ in order to obtain $\mathsf{f}_1(\mathsf{a}_1, \mathsf{b}_1) \to^* 1$. This however gives rise to further violations of compatibility, namely $\mathsf{f}_1(\mathsf{a}_1, \mathsf{a}_1) \to_{\mathsf{match}(\mathcal{R})} \mathsf{a}_2$ with $\mathsf{f}_1(\mathsf{a}_1, \mathsf{a}_1) \to^* 1$, $\mathsf{f}_1(\mathsf{b}_1, \mathsf{b}_1) \to_{\mathsf{match}(\mathcal{R})} \mathsf{b}_2$ with $\mathsf{f}_1(\mathsf{b}_1, \mathsf{b}_1) \to^* 1$, and $\mathsf{f}_1(x, x) \to_{\mathsf{match}(\mathcal{R})} \mathsf{f}_2(\mathsf{a}_2, \mathsf{b}_2)$ with $\mathsf{f}_1(1, 1) \to 1$. To solve the first two violations the transitions $\mathsf{a}_2 \to 1$ and $\mathsf{b}_2 \to 1$ have to be added. Afterwards the tree automaton consist of the following transitions:

  $$\begin{array}{rrr}
  \mathsf{a}_0 \to 1 & \mathsf{b}_0 \to 1 & \mathsf{f}_0(1,1) \to 1 \\
  \mathsf{a}_1 \to 1 & \mathsf{b}_1 \to 1 & \mathsf{f}_1(1,1) \to 1 \\
  \mathsf{a}_2 \to 1 & \mathsf{b}_2 \to 1 &
  \end{array}$$

  It is easy to see that the new situation is similar to the one at the beginning: we have to establish $\mathsf{f}_2(\mathsf{a}_2, \mathsf{b}_2) \to^* 1$ and may reuse one or both of the transitions $\mathsf{a}_2 \to 1$ and $\mathsf{b}_2 \to 1$.

- Suppose we reuse $\mathsf{a}_1 \to 1$ but not $\mathsf{b}_1 \to 1$. That means we have to add a new state 2 and transitions $\mathsf{b}_1 \to 2$ and $\mathsf{f}_1(1, 2) \to 1$ resulting in the following transitions:

  $$\begin{array}{rrr}
  \mathsf{a}_0 \to 1 & \mathsf{b}_0 \to 1 & \mathsf{f}_0(1,1) \to 1 \\
  \mathsf{a}_1 \to 1 & \mathsf{b}_1 \to 1 \mid 2 & \mathsf{f}_1(1,2) \to 1
  \end{array}$$

  Making these transitions deterministic produces an automaton that includes $\mathsf{b}_0 \to 1$, $\mathsf{f}_0(1, 1) \to 1$, and $\mathsf{b}_1 \to \{1, 2\}$. To simplify the presentation we identify states $\{q\}$ with $q \in \{1,2\}$ by q. Because the transition $\mathsf{b}_1 \to 1$ was removed, the second violation of compatibility that we considered, $\mathsf{f}_0(\mathsf{b}_0,\mathsf{b}_0) \to_{\mathsf{match}(\mathcal{R})} \mathsf{b}_1$ and $\mathsf{f}_0(\mathsf{b}_0,\mathsf{b}_0) \to 1$, reappears. So we have to add $\mathsf{b}_1 \to 1$ again, but each time we make the automaton deterministic this transition is deleted.

- The remaining options would be to choose a fresh state for a_1 or for both a_1 and b_1. However they all give rise to the same situation.

So by using deterministic automata we will never achieve compatibility. The problem is clearly the removal of transitions that were added in an earlier stage to ensure compatibility and that is precisely the reason why we use quasi-deterministic tree automata.

Assume that we have constructed a quasi-deterministic tree automaton $\mathcal{A}$ that is compatible with $e(\mathcal{R})$ and $\mathsf{lift}_0(L)$. To infer that the TRS $\mathcal{R}$ is e-raise-bounded for L, it must be guaranteed that $\mathcal{A}$ accepts at least $\xrightarrow{\mathsf{r}}{}^{*}_{e(\mathcal{R})}(\mathsf{lift}_0(L))$. From Theorem 3.16 we know that compatibility of the tree automaton $\mathcal{A}$ yields $\to^{*}_{e(\mathcal{R})}(\mathsf{lift}_0(L)) \subseteq \mathcal{L}(\mathcal{A})$ for any TRS $\mathcal{R}$. However that is not enough to conclude e-raise-boundedness. We also have to ensure that $\mathcal{A}$ is closed under the implicit raise-steps caused by the rewrite relation $\xrightarrow{\mathsf{r}}$. How this can be done automatically is explained in the next subsection.

4.4.2 Raise-Consistent Tree Automata

A naive (and sound) approach to guarantee that the implicit raise-rules in the definition of $\xrightarrow{\mathsf{r}}$ are taken into account would be to require compatibility with all raise-rules $f_c(x_1,\dots,x_n) \to f_{c+1}(x_1,\dots,x_n)$ for which f_d with $d \geqslant c+1$ appears in the current set of transitions. However, the following example shows that this approach may over-approximate the essential raise-steps too much.

Example 4.20. Let us continue Example 4.19. We have $\mathsf{f}_0(x,y) \to_{\mathsf{raise}(\mathcal{F})} \mathsf{f}_1(x,y)$ with $\mathsf{f}_0(1,1) \to 1$. Compatibility requires the addition of the transition $\mathsf{f}_1(1,1) \to 1$, causing a new compatibility violation $\mathsf{f}_1(x,x) \to_{\mathsf{match}(\mathcal{R})} \mathsf{f}_2(\mathsf{a}_2,\mathsf{b}_2)$ with $\mathsf{f}_1(1,1) \to 1$. After establishing the path $\mathsf{f}_2(\mathsf{a}_2,\mathsf{b}_2) \to^{*} 1$, f_2 will make its appearance and thus we have to consider $\mathsf{f}_1(x,y) \to_{\mathsf{raise}(\mathcal{F})} \mathsf{f}_2(x,y)$ with $\mathsf{f}_1(1,1) \to 1$. This yields the transition $\mathsf{f}_2(1,1) \to 1$. Clearly, this process will not terminate.

To avoid the behavior in the previous example, we now outline a better approach to handle raise-rules. Let $f_c(q_1,\dots,q_n) \to q$ be a transition that we add to the current set Δ of transitions, either to resolve a compatibility violation or to satisfy the quasi-determinism condition. Then, for every transition $f_d(q_1,\dots,q_n) \to p \in \Delta$ with $d < c$ we add $f_c(q_1,\dots,q_n) \to p$ to Δ and for every transition $f_d(q_1,\dots,q_n) \to p \in \Delta$ with $d > c$ we add $f_d(q_1,\dots,q_n) \to q$ to Δ. The automata resulting from this implicit handling of raise-rules satisfy the property defined below.

Definition 4.21. Let $\mathcal{A} = (\mathcal{F}_N, Q, Q_f, \Delta)$ be a tree automaton with N a finite subset of $\mathbb{N}$. We say that $\mathcal{A}$ is *raise-consistent* if for every transition

$f_c(q_1, \ldots, q_n) \to q \in \Delta$ and left-hand side $f_d(q_1, \ldots, q_n) \in \mathsf{lhs}(\Delta)$ with $c < d$, the transition $f_d(q_1, \ldots, q_n) \to q$ belongs to Δ.

Let us illustrate the above definition on an example.

Example 4.22. The tree automaton $\mathcal{A}$ consisting of the final state 1 and the transitions

$$\mathsf{a}_0 \to 1 \qquad \mathsf{a}_1 \to 1 \mid 2 \qquad \mathsf{f}_0(1,2) \to 1 \qquad \mathsf{f}_2(1,2) \to 2$$

is not raise-consistent because $\mathsf{f}_0(1,2) \to 1$ but not $\mathsf{f}_2(1,2) \to 1$. Adding the latter transition to $\mathcal{A}$ makes it raise-consistent.

In the remainder of the subsection we show that by constructing a quasi-deterministic and raise-consistent tree automaton $\mathcal{A}$ that is compatible with $e(\mathcal{R})$ and $\mathsf{lift}_0(L)$ it is guaranteed that $\mathcal{A}$ accepts $\xrightarrow{\mathsf{r}}^*_{e(\mathcal{R})}(\mathsf{lift}_0(L))$. We start by proving the following technical lemma, which expresses a key property of raise-consistent tree automata.

Lemma 4.23. *Let $\mathcal{A} = (\mathcal{F}_N, Q, Q_f, \Delta)$ be a quasi-deterministic tree automaton with N a finite subset of $\mathbb{N}$. If $\mathcal{A}$ is raise-consistent then for all ground terms $s, t \in \mathcal{T}(\mathcal{F}_N)$ and states $p_s, q_t \in Q$ with $\mathsf{base}(s) = \mathsf{base}(t)$, $s \to^*_{\Delta_\phi} p_s$, and $t \to^*_{\Delta_\phi} p_t$ there exists a state $q \in Q$ such that $s \uparrow t \to^*_{\Delta_\phi} q$, $q \succeq_\phi p_s$, and $q \succeq_\phi p_t$.*

Proof. We prove the lemma by induction on the structure of s and t. If s and t are constants then $s \uparrow t \in \{s, t\}$. If $t \succeq s$ then $s \uparrow t = t$ and $p_s \in Q(t)$ by the definition of raise-consistency. Likewise, if $s \succeq t$ then $s \uparrow t = s$ and $p_t \in Q(s)$ for the same reason. So by taking in both cases $q = \phi(s \uparrow t)$ it follows that $q \succeq_\phi p_s$ and $q \succeq_\phi p_t$ according to Definition 3.10. For the induction step suppose that $s = f_j(s_1, \ldots, s_n)$ and $t = f_k(t_1, \ldots, t_n)$ with $s \to^*_{\Delta_\phi} f_j(p_{s_1}, \ldots, p_{s_n}) \to_{\Delta_\phi} p_s$ and $t \to^*_{\Delta_\phi} f_k(p_{t_1}, \ldots, p_{t_n}) \to_{\Delta_\phi} p_t$. The induction hypothesis yields for every $i \in \{1, \ldots, n\}$ a state $q_i \in Q$ such that $s_i \uparrow t_i \to^*_{\Delta_\phi} q_i$ with $q_i \succeq_\phi p_{s_i}$ and $q_i \succeq_\phi p_{t_i}$. Let $C_i = f_j(s_1 \uparrow t_1, \ldots, s_{i-1} \uparrow t_{i-1}, \Box, s_{i+1}, \ldots, s_n)$ and $D_i = f_k(s_1 \uparrow t_1, \ldots, s_{i-1} \uparrow t_{i-1}, \Box, t_{i+1}, \ldots, t_n)$ for all $i \in \{1, \ldots, n\}$. Let $v_0 = p_s$ and $w_0 = p_t$. Since $C_1[p_{s_1}] \to^+_{\Delta_\phi} v_0$ and $q_1 \succeq_\phi p_{s_1}$, Lemma 3.15 yields a state $v_1 \in Q$ such that $C_1[q_1] \to^+_{\Delta_\phi} v_1$ and $v_1 \succeq_\phi v_0$. Likewise, $D_1[p_{t_1}] \to^+_{\Delta_\phi} w_0$ with $q_1 \succeq_\phi p_{t_1}$ yields a state $w_1 \in Q$ such that $D_1[q_1] \to^+_{\Delta_\phi} w_1$ and $w_1 \succeq_\phi w_0$. Repeating this argumentation $n - 1$ times produces states $v_i, w_i \in Q$ such that $C_i[q_i] \to^+_{\Delta_\phi} v_i$ with $v_i \succeq_\phi v_{i-1}$ and $D_i[q_i] \to^+_{\Delta_\phi} w_i$ with $w_i \succeq_\phi w_{i-1}$ for all $i \in \{1, \ldots, n\}$. It follows that $f_j(q_1, \ldots, q_n) \to_{\Delta_\phi} v_n$ and $f_k(q_1, \ldots, q_n) \to_{\Delta_\phi} w_n$. Furthermore, by the transitivity of $\succeq_\phi$ we obtain $v_n \succeq_\phi v_0$ and $w_n \succeq_\phi w_0$. Now let $m = \max\{j, k\}$ and $l = f_m(q_1, \ldots, q_n)$. Raise-consistency of $\mathcal{A}$ yields $v_n, w_n \in Q(l)$. Together with the fact that $\mathcal{A}$ is quasi-deterministic we obtain $s \uparrow t = f_m(s_1 \uparrow t_1, \ldots, s_n \uparrow t_n) \to^*_{\Delta_\phi} l \to_{\Delta_\phi} \phi(l)$, $\phi(l) \succeq_\phi v_n$, and $\phi(l) \succeq_\phi w_n$. So by taking $q = \phi(l)$ we obtain $s \uparrow t \to^*_{\Delta_\phi} q$, $q \succeq_\phi p_s$, and $q \succeq_\phi p_t$ as desired. Note that the last two properties follow from the transitivity of $\succeq_\phi$. $\Box$

Using the above lemma we are now ready to show that raise-consistency suffice to handle the implicit raise-steps caused by the rewrite relation $\xrightarrow{\mathsf{r}}$.

Theorem 4.24. *Let $\mathcal{R}$ be a TRS and L a language. Let $\mathcal{A}$ be a raise-consistent and quasi-deterministic tree automaton. If $\mathcal{A}$ is compatible with $e(\mathcal{R})$ and $\mathsf{lift}_0(L)$ then $\mathcal{R}$ is e-raise-bounded for L.*

Proof. Let $\mathcal{F}$ be the signature of $\mathcal{R}$ and $\mathcal{A} = (\mathcal{F}_N, Q, Q_f, \Delta)$ for some finite subset N of $\mathbb{N}$. We have $\mathsf{lift}_0(L) \subseteq \mathcal{L}(\mathcal{A})$. Let $s \in \mathcal{L}(\mathcal{A})$ and $s \xrightarrow{\mathsf{r}}_{l \to r} t$ with $l \to r \in e(\mathcal{R})$. Then there is a term s' such that $s \to^*_{\mathsf{raise}(\mathcal{F})} s' \to_{l \to r} t$. We show that $s' \in \mathcal{L}(\mathcal{A})$. If l is linear then $s = s'$ and we are done. Suppose l is non-linear. To simplify the notation we assume that $l = f(x,x)$. Let p be the position at which the rewrite rule $l \to r$ is applied. We may write $s = s[f(s_1,s_2)]_p$ and $s' = s[f(u,u)]_p$ with $\mathsf{base}(s_1) = \mathsf{base}(s_2)$ and $u = s_1 \uparrow s_2$. Since s belongs to $\mathcal{L}(\mathcal{A})$, there exist states $p_{s_1}, p_{s_2}, q \in Q$ and $q_s \in Q_f$ such that $s \to^*_{\Delta_\phi} s[f(p_{s_1},p_{s_2})]_p \to_{\Delta_\phi} s[q]_p \to^*_{\Delta_\phi} q_s$. The previous lemma yields a state $p_u \in Q$ such that $u \to^*_{\Delta_\phi} p_u$, $p_u \succeq_\phi p_{s_1}$, and $p_u \succeq_\phi p_{s_2}$. Since $p_u \succeq_\phi p_{s_1}$ and $p_u \succeq_\phi p_{s_2}$ we know that p_u subsumes p_{s_1} and p_{s_2}. It follows that there are states $q', q'' \in Q$ such that $f(p_u, p_{s_2}) \to q'$ and $f(p_u,p_u) \to q''$ belong to Δ. According to Definition 3.10 we have $q' \succeq_\phi q$ and $q'' \succeq_\phi q'$. Transitivity of $\succeq_\phi$ yields $q'' \succeq_\phi q$. Because $s[q]_p \to^*_{\Delta_\phi} q_s$ and $q'' \succeq_\phi q$, Lemma 3.15 yields a state $q_{s'} \in Q$ such that $s[q'']_p \to^*_{\Delta_\phi} q_{s'}$ and $q_{s'} \succeq_\phi q_s$. Putting things together produces the derivation $s' = s[f(u,u)]_p \to^*_{\Delta_\phi} s[f(p_u,p_u)]_p \to_{\Delta_\phi} s[q'']_p \to^*_{\Delta_\phi} q_{s'}$. Because $q_{s'}$ subsumes q_s we know that $q_{s'} \in Q_f$ and hence $s' \in \mathcal{L}(\mathcal{A})$. Now that $s' \in \mathcal{L}(\mathcal{A})$ is established, we obtain $t \in \mathcal{L}(\mathcal{A})$ from the compatibility of $\mathcal{A}$ and $e(\mathcal{R})$, as in the proof of Theorem 3.16. □

Example 4.25. Since the final quasi-deterministic tree automaton in Example 4.19 is raise-consistent and compatible with $\mathsf{match}(\mathcal{R})$ and $\mathsf{lift}_0(\mathcal{T}(\mathcal{F}))$, $\mathcal{R}$ is match-raise-bounded for $\mathcal{T}(\mathcal{F})$ by Theorem 4.24.

4.4.3 Quasi-Compatible Tree Automata

By using the explicit approach for handling raise-rules described in the first paragraph of Subsection 4.4.2 or the implicit approach using raise-consistent tree automata, it is often the case that a transition is duplicated by increasing the height of the function symbol of the left-hand side. As soon as this happens, the transition with the smaller height is in principle useless since in each further compatibility violation the new transition with the greater height can be used instead. To be able to simplify tree automata by removing such transitions we introduce so called *quasi-compatible* tree automata.

Definition 4.26. Let $\mathcal{R}$ be a non-left-linear TRS and L a language. Let $\mathcal{A} = (\mathcal{F}_N, Q, Q_f, \Delta)$ be a quasi-deterministic tree automaton with N a finite subset of $\mathbb{N}$. We say that $\mathcal{A}$ is *quasi-compatible* with $\mathcal{R}$ and L if for all $t \in L$ there is a term $t' \in \mathcal{L}(\mathcal{A})$ such that $t' \succeq t$ and for each rewrite rule $l \to r \in \mathcal{R}$ and state substitution $\sigma\colon \mathcal{V}\mathsf{ar}(l) \to Q_\phi$ such that $l\sigma \to^*_{\Delta_\phi} q$ it holds that $r'\sigma \to^*_\Delta q$ for some $r' \succeq r$.

In the following we show that each quasi-deterministic and raise-consistent tree automaton $\mathcal{A}$ that is quasi-compatible with $e(\mathcal{R})$ and $\mathsf{lift}_0(L)$ can be transformed into a quasi-deterministic and raise-consistent tree automaton that is compatible with $e(\mathcal{R})$ and $\mathsf{lift}_0(L)$. As an immediate consequence we obtain that $\mathcal{R}$ is e-raise-bounded for L if $\mathcal{A}$ is quasi-compatible with $e(\mathcal{R})$ and $\mathsf{lift}_0(L)$. To perform this transformation we need the notion defined below.

Definition 4.27. Let $\mathcal{A} = (\mathcal{F}_N, Q, Q_f, \Delta)$ be a tree automaton with N a finite subset of $\mathbb{N}$. We say that $\mathcal{A}$ is *height-complete* if for all $f_i(q_1, \ldots, q_n) \to q \in \Delta$ we have $f_j(q_1, \ldots, q_n) \to q \in \Delta$ for all $j \in \{0, \ldots, i-1\}$.

First we show that every height-complete extension of a tree automaton $\mathcal{A}$ is raise-consistent and quasi-deterministic if $\mathcal{A}$ is raise-consistent and quasi-deterministic.

Lemma 4.28. *Let $\mathcal{A} = (\mathcal{F}_N, Q, Q_f, \Delta)$ be a raise-consistent and quasi-deterministic tree automaton with N a finite subset of $\mathbb{N}$. Let $\mathcal{A}' = (\mathcal{F}_{N'}, Q, Q_f, \Delta')$ be the smallest height-complete tree automaton such that $N \subseteq N'$ and $\Delta \subseteq \Delta'$. Then $\mathcal{A}'$ is raise-consistent and quasi-deterministic.*

Proof. To simplify the presentation, let $Q_{\mathcal{A}'}(l)$ be denoted by $Q'(l)$ and let $\phi_{\mathcal{A}}$ and $\phi_{\mathcal{A}'}$ be abbreviated by ϕ and ϕ'. It is easy to see that the raise-consistency of $\mathcal{A}'$ is an immediate consequence of Definitions 4.21 and 4.27. Hence we only have to show that $\mathcal{A}'$ is quasi-deterministic. To this end, let ϕ be defined as $\phi'(l) = \phi(l')$ for all $l \in \mathsf{lhs}(\Delta')$ where $l' \succeq l$ denotes the left-hand side in $\mathsf{lhs}(\Delta)$ such that $l'(\epsilon)$ is maximal, that is, $l'(\epsilon) \succeq l''(\epsilon)$ for all $l'' \in \mathsf{lhs}(\Delta)$ with $\mathsf{base}(l'') = \mathsf{base}(l')$. Height-consistency of $\mathcal{A}'$ yields that $Q'(l) = Q'(l')$ for all $l, l' \in \mathsf{lhs}(\Delta')$ with $l' \succeq l$. Hence $\phi'(l) \in Q'(l)$ for all $l \in \mathsf{lhs}(\Delta')$ as required by Definition 3.10. It remains to show that p subsumes q with respect to Δ' whenever $p \succeq_{\phi'} q$ with $p, q \in Q'$. To prove this property of $\mathcal{A}'$ we first show that $p \succeq_{\phi} q$ whenever $p \succeq_{\phi'} q$. Assume that $p \succeq_{\phi'} q$ with $p = \phi'(l)$ and $q \in Q'(l)$ for some $l \in \mathsf{lhs}(\Delta')$. Because $\mathcal{A}'$ is the smallest height-complete extension of $\mathcal{A}$ we know that there is a left-hand side $l' \in \mathsf{lhs}(\Delta)$ such that $l' \succeq l$ and $Q(l') = Q'(l)$. Because $\phi'(l) = \phi(l')$ we have $p \succeq_{\phi} q$ according to Definition 3.10. It remains to show that $p \succeq_{\phi} q$ if $p = \phi'(u)$ and $q = \phi'(v)$ for arbitrary left-hand sides $u = f(p_1, \ldots, p_{i-1}, p', p_{i+1}, \ldots, p_n)$ and $v = f(p_1, \ldots, p_{i-1}, q', p_{i+1}, \ldots, p_n)$ in $\mathsf{lhs}(\Delta')$ with $p' \succeq_{\phi} q'$. The desired result follows then by induction and transitivity of $\succeq_{\phi}$. As before the height-completeness of $\mathcal{A}$ yields two left-hand sides $u', v' \in \mathsf{lhs}(\Delta)$ such that $u' \succeq u$, $v' \succeq v$, $Q(u') = Q'(u)$, and $Q(v') = Q'(v)$. Because $p' \succeq_{\phi} q'$, $\phi'(u) = \phi(u')$, and $\phi'(v) = \phi(v')$ we know that $p \succeq_{\phi} q$ according to Definition 3.10. This concludes the proof of the statement.

Now assume that there are states $p, q \in Q'$ such that $p \succeq_{\phi'} q$ but p does not subsume q with respect to Δ'. According to Definition 3.10 there is a left-hand side $u = f(p_1, \ldots, p_{i-1}, p, p_{i+1}, \ldots, p_n)$ in $\mathsf{lhs}(\Delta')$ such that the term $v = f(p_1, \ldots, p_{i-1}, q, p_{i+1}, \ldots, p_n)$ does not belong to $\mathsf{lhs}(\Delta')$. Because $\mathcal{A}'$ is the smallest height-complete extension of $\mathcal{A}$ we know that there is a left-hand side $u' \in \mathsf{lhs}(\Delta)$ such that $u' \succeq u$. Since $\mathcal{A}$ is quasi-deterministic and $p \succeq_{\phi} q$ by the previous statement we know that p subsumes q with respect to Δ. Hence there is a left-hand side $v' \in \mathsf{lhs}(\Delta)$ such that $v' \succeq v$. Height-consistency yields $v \in \mathsf{lhs}(\Delta')$ which contradicts our assumption. $\square$

Next we show that quasi-compatibility implies e-raise-boundedness and hence the termination of the considered TRS.

Theorem 4.29. *Let $\mathcal{R}$ be a TRS and L a language. Let $\mathcal{A}$ be a raise-consistent and quasi-deterministic tree automaton. If $\mathcal{A}$ is quasi-compatible with $e(\mathcal{R})$ and $\mathsf{lift}_0(L)$ then $\mathcal{R}$ is e-raise-bounded for L.*

Proof. To simplify the presentation we abbreviate $\phi_{\mathcal{A}}$ by ϕ and $\phi_{\mathcal{A}'}$ by ϕ'. Let $\mathcal{A} = (\mathcal{F}_N, Q, Q_f, \Delta)$ for some finite set $N \subset \mathbb{N}$ and let $\mathcal{A}' = (\mathcal{F}_{N'}, Q, Q_f, \Delta')$ be the smallest height-complete tree-automaton such that $N \subseteq N'$ and $\Delta \subseteq \Delta'$. Due to Lemma 4.28, $\mathcal{A}'$ is quasi-deterministic and raise-consistent. We show that $\mathcal{A}'$ is compatible with $e(\mathcal{R})$ and $\mathsf{lift}_0(L)$. Assume to the contrary that this does not hold. Then there is a rewrite rule $l \to r \in e(\mathcal{R})$ and a state substitution $\sigma\colon \mathcal{V}\mathsf{ar}(l) \to Q_{\phi'}$ such that $l\sigma \to^*_{\Delta'_{\phi'}} q$ but not $r\sigma \to^*_{\Delta'} q$. By the construction of $\mathcal{A}'$ there exists a term $l' \succeq l$ such that $l'\sigma \to^*_{\Delta_\phi} q$. Let $r' \succeq r$ such that $l' \to r' \in e(\mathcal{R})$. Since $\mathcal{A}$ is quasi-compatible with $e(\mathcal{R})$ and $\mathsf{lift}_0(L)$ there must be a term $r'' \succeq r'$ such that $r''\sigma \to^*_\Delta q$ and thus also $r''\sigma \to^*_{\Delta'} q$. Let $c \in N'$ such that $\mathsf{lift}_c(\mathsf{base}(r)) = r$ and let $l_1 \to p_1, \ldots, l_n \to p_n$ be the transitions in Δ' which are used in the derivation $r''\sigma \to^*_{\Delta'} q$. From $r'' \succeq r$ and the height-completeness of $\mathcal{A}'$ we infer that $\mathsf{lift}_c(\mathsf{base}(l_i)) \to p_i \in \Delta'$ for all $i \in \{1, \ldots, n\}$. Hence $r\sigma \to^*_{\Delta'} q$, contradicting our assumption. Putting things together yields that $\mathcal{R}$ is e-raise-bounded for L according to Theorem 4.24. □

Because e-raise-boundedness coincides with e-boundedness for left-linear rewrite systems it is obvious that quasi-compatible tree automata can also be used to verify e-bounds. In this particular case the definition of quasi-compatible tree automata can be slightly simplified.

Definition 4.30. Let $\mathcal{R}$ be some left-linear TRS and L a language. Let $\mathcal{A} = (\mathcal{F}_N, Q, Q_f, \Delta)$ be a tree automaton with N a finite subset of $\mathbb{N}$. We say that $\mathcal{A}$ is *quasi-compatible* with $\mathcal{R}$ and L if for all $t \in L$ there is a term $t' \in \mathcal{L}(\mathcal{A})$ such that $t' \succeq t$ and for each rewrite rule $l \to r \in \mathcal{R}$ and state substitution $\sigma\colon \mathcal{V}\mathsf{ar}(l) \to Q$ such that $l\sigma \to^*_\Delta q$ it holds that $r'\sigma \to^*_\Delta q$ for some $r' \succeq r$.

Using the above definition it is straightforward to prove that quasi-compatible tree automata can be used to verify e-boundedness.

Theorem 4.31. *Let $\mathcal{R}$ be a left-linear TRS and L a language. If there is a tree automaton $\mathcal{A}$ which is quasi-compatible with $e(\mathcal{R})$ and $\mathsf{lift}_0(L)$ then $\mathcal{R}$ is e-bounded for L.*

Proof. Analogues to the proof of Theorem 4.29 using Theorem 4.17 instead of Theorem 4.24. □

The general idea for constructing a (quasi-deterministic and raise-consistent) tree automaton that is quasi-compatible with a TRS $e(\mathcal{R})$ and a language $\mathsf{lift}_0(L)$ is quite similar to the procedure described in Section 3.1 (Section 3.4) for constructing a compatible (and quasi-deterministic) tree automaton. However, instead of checking for compatibility violations we look for violations of the quasi-compatibility requirement: $l\sigma \to^*_\Delta q$ ($l\sigma \to^*_{\Delta_\phi} q$) for some rewrite rule $l \to r$, state substitution $\sigma\colon \mathcal{V}\mathsf{ar}(l) \to Q$ ($\sigma\colon \mathcal{V}\mathsf{ar}(l) \to Q_\phi$), and state q, but not $r'\sigma \to^*_\Delta q$ for any $r' \succeq r$. After that the path $r\sigma \to^*_\Delta q$ has to be established by adding new states and transitions to the current automaton. Finally, to benefit from the use of quasi-compatible tree automata we delete all transitions $f_c(p_1,\dots,p_n) \to p$ for which there is a base-equivalent transition $f_d(p_1,\dots,p_n) \to p$ with $d > c$. This process is repeated until a (quasi-deterministic, raise-consistent, and) quasi-compatible tree automaton is obtained. To ensure that the constructed tree automaton is raise-consistent and hence closed under the implicit raise-steps caused by the rewrite relation $\xrightarrow{\mathsf{r}}$ if $\mathcal{R}$ is not left-linear, one of the procedures presented in Subsection 4.4.2 can be applied.

Example 4.32. A quasi-deterministic and raise-consistent tree automaton that is quasi-compatible with the TRS $\mathsf{match}(\mathcal{R})$ of Example 4.19 consists of the transitions

$$\begin{array}{llll}
\mathsf{a}_1 \to 1 \mid 2 \mid 4 & \mathsf{b}_1 \to 1 \mid 3 \mid 5 & & \\
\mathsf{f}_0(1,1) \to 1 & \mathsf{f}_0(1,4) \to 1 & \mathsf{f}_0(1,5) \to 1 & \mathsf{f}_0(4,1) \to 1 \\
\mathsf{f}_0(4,4) \to 1 & \mathsf{f}_0(5,1) \to 1 & \mathsf{f}_0(5,4) \to 1 & \mathsf{f}_0(5,5) \to 1 \\
\mathsf{f}_1(2,3) \to 1 & \mathsf{f}_1(2,5) \to 1 & \mathsf{f}_1(4,3) \to 1 & \mathsf{f}_1(4,5) \to 1
\end{array}$$

and the final states 1, 4, and 5. With respect to the quasi-deterministic, raise-consistent, and compatible tree automaton given in Example 4.19, the transitions $\mathsf{a}_0 \to 1$, $\mathsf{b}_0 \to 1$, and $\mathsf{f}_0(4,5) \to 1$ are removed. So by constructing a quasi-compatible tree automaton a slightly smaller automaton is obtained.

Example 4.33. The tree automaton consisting of the transitions

$$\mathsf{a}_0 \to 1 \qquad \mathsf{g}_0(1) \to 1 \qquad \mathsf{f}_0(1,1) \to 1 \qquad \mathsf{f}_1(1,1) \to 1$$

and the final state 1 is compatible with the TRS $\mathsf{match}(\mathcal{R})$ and the language $\mathsf{lift}_0(\mathcal{T}(\mathcal{F}))$. Here $\mathcal{R}$ is the TRS of Example 4.6 over the signature $\mathcal{F} = \{\mathsf{a},\mathsf{f},\mathsf{g}\}$. If we construct a quasi-compatible tree automaton instead of a compatible tree automaton then the transition $\mathsf{f}_0(1,1) \to 1$ is removed in step one of the completion procedure because the transition $\mathsf{f}_1(1,1) \to 1$ is added in order to solve some compatibility violation. As a result the obtained quasi-compatible tree automaton consists of 3 instead of 4 transitions.

4.5 Forward Closures

When proving the termination of a TRS $\mathcal{R}$ that is non-overlapping [25] or right-linear [7] it is sufficient to restrict attention to the set $\mathsf{RFC}_{\mathsf{rhs}(\mathcal{R})}(\mathcal{R})$ of *right-hand sides of forward closures*. This set is defined as the closure of the right-hand sides of the rules in $\mathcal{R}$ under narrowing.

Definition 4.34. Let $\mathcal{R}$ be a TRS and L a language. The set $\mathsf{RFC}_L(\mathcal{R})$ is the least extension of L such that $t[r]_p\sigma \in \mathsf{RFC}_L(\mathcal{R})$ whenever $t \in \mathsf{RFC}_L(\mathcal{R})$ and there exist a position $p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(t)$ and a fresh variant $l \to r$ of a rewrite rule in $\mathcal{R}$ with σ a most general unifier of $t|_p$ and l. If L is a singleton set consisting of a term t we write $\mathsf{RFC}_t(\mathcal{R})$ instead of $\mathsf{RFC}_{\{t\}}(\mathcal{R})$.

Dershowitz [7] obtained the following result.

Theorem 4.35. *A right-linear TRS $\mathcal{R}$ is terminating if and only if $\mathcal{R}$ is terminating on $\mathsf{RFC}_{\mathsf{rhs}(\mathcal{R})}(\mathcal{R})$.* $\square$

The following concept has been introduced in [24]. It enables the simulation of narrowing in the definition of right-hand sides of forward closures by rewriting. This makes it possible to use tree automata to compute an approximation of $\mathsf{RFC}_L(\mathcal{R})$ for linear $\mathcal{R}$.

Definition 4.36. Let $\mathcal{R}$ be a TRS and $\#$ a fresh function symbol. The TRS $\mathcal{R}_\#$ is defined as the least extension of $\mathcal{R}$ that is closed under the following operation. If $l \to r \in \mathcal{R}_\#$ and $p \in \mathcal{P}\mathsf{os}_\mathcal{F}(l) \setminus \{\epsilon\}$ then $l[\#]_p \to r\sigma \in \mathcal{R}_\#$. Here the substitution σ is defined by $\sigma(x) = \#$ if $x \in \mathcal{V}\mathsf{ar}(l|_p)$ and $\sigma(x) = x$ otherwise. The substitution that maps all variables to $\#$ is denoted by $\sigma_\#$.

The following results are proved in [24].

Lemma 4.37. *Let $\mathcal{R}$ be a linear TRS and L a set of linear terms. We have $\mathsf{RFC}_L(\mathcal{R})\sigma_\# = \to^*_{\mathcal{R}_\#}(L\sigma_\#)$.* $\square$

Theorem 4.38. *If a linear TRS $\mathcal{R}$ is match-bounded for $\to^*_{\mathcal{R}_\#}(\mathsf{rhs}(\mathcal{R})\sigma_\#)$ then $\mathcal{R}$ is terminating.* $\square$

Let us illustrate the above results on an example.

Example 4.39. For the TRS $\mathcal{R}$ of Example 4.2, $\mathcal{R}_\#$ consists of the following rewrite rules:

$$\mathsf{f}(\mathsf{g}(x,\mathsf{h}(y))) \to \mathsf{g}(\mathsf{h}(\mathsf{f}(x)),y) \qquad \mathsf{f}(\mathsf{g}(x,\#)) \to \mathsf{g}(\mathsf{h}(\mathsf{f}(x)),\#)$$
$$\mathsf{f}(\#) \to \mathsf{g}(\mathsf{h}(\mathsf{f}(\#)),\#)$$

By using these rewrite rules we can now easily compute the set $\mathsf{RFC}_{\mathsf{rhs}(\mathcal{R})}(\mathcal{R})\sigma_\#$. We have

$$\begin{aligned}\mathsf{g}(\mathsf{h}(\mathsf{f}(\#)),\#) &\to_{\mathcal{R}_\#} \mathsf{g}(\mathsf{h}(\mathsf{g}(\mathsf{h}(\mathsf{f}(\#)),\#)),\#)\\ &\to_{\mathcal{R}_\#} \mathsf{g}(\mathsf{h}(\mathsf{g}(\mathsf{h}(\mathsf{g}(\mathsf{h}(\mathsf{f}(\#)),\#)),\#)),\#)\\ &\to_{\mathcal{R}_\#} \mathsf{g}(\mathsf{h}(\mathsf{g}(\mathsf{h}(\mathsf{g}(\mathsf{h}(\mathsf{g}(\mathsf{h}(\mathsf{f}(\#)),\#)),\#)),\#)),\#)\\ &\to_{\mathcal{R}_\#} \cdots\end{aligned}$$

and hence $\mathsf{RFC}_{\mathsf{rhs}(\mathcal{R})}(\mathcal{R})\sigma_\# = \to^*_{\mathcal{R}_\#}(\mathsf{rhs}(\mathcal{R})\sigma_\#) = \{\mathsf{g}(\mathsf{h}(x),\#)\sigma^n\tau \mid n \geqslant 0\}$ where $\sigma = \{x \mapsto \mathsf{g}(\mathsf{h}(x),\#)\}$ and $\tau = \{x \mapsto \mathsf{f}(\#)\}$. Since all terms in the set $\to^*_{\mathcal{R}_\#}(\mathsf{rhs}(\mathcal{R})\sigma_\#)$ are in normal form with respect to $\mathcal{R}$, it is obvious that $\mathcal{R}$ is match-bounded for $\to^*_{\mathcal{R}_\#}(\mathsf{rhs}(\mathcal{R})\sigma_\#)$ by 0 and hence terminating according to Theorem 4.38. At the end of this section we explain in detail how this can be automatically checked.

In order to obtain corresponding results for arbitrary right-linear TRSs, we linearize left-hand sides of rewrite rules.

Definition 4.40. Let t be a term. The set of linear terms s, with $\mathcal{V}\mathsf{ar}(t) \subseteq \mathcal{V}\mathsf{ar}(s)$, for which there exists a variable substitution $\tau\colon \mathcal{V}\mathsf{ar}(s) \setminus \mathcal{V}\mathsf{ar}(t) \to \mathcal{V}\mathsf{ar}(t)$ such that $s\tau = t$ is denoted by $\mathsf{linear}(t)$. Let $\mathcal{R}$ be a TRS. The set of rewrite rules $\{l' \to r \mid l \to r \in \mathcal{R}$ and $l' \in \mathsf{linear}(l)\}$ is denoted by $\mathsf{linear}(\mathcal{R})$.

In the following we write $\mathcal{R}'_\#$ for $\mathsf{linear}(\mathcal{R})_\#$. In general $\mathsf{linear}(\mathcal{R})$ and hence $\mathcal{R}'_\#$ consists of infinitely many rewrite rules since variables in $\mathcal{V}\mathsf{ar}(l') \setminus \mathcal{V}\mathsf{ar}(l)$ are not constrained. When using $\mathcal{R}'_\#$ to approximate $\mathsf{RFC}_L(\mathcal{R})\sigma_\#$ it is enough to consider a finite subset of $\mathcal{R}'_\#$ which ignores different variants of rules. Note that in this case $\mathcal{R}_\# = \mathcal{R}'_\#$ for linear TRSs $\mathcal{R}$.

Example 4.41. The TRS $\mathsf{linear}(\mathcal{R})$, with $\mathcal{R}$ consisting of the rewrite rules

$$\mathsf{f}(x,x) \to \mathsf{f}(\mathsf{h}(x),\mathsf{a}) \qquad \mathsf{f}(\mathsf{h}(x),x) \to \mathsf{g}(x) \qquad \mathsf{g}(x) \to \mathsf{f}(x,\mathsf{a})$$

contains the following rules:

$$\begin{array}{lll} \mathsf{f}(x',x) \to \mathsf{f}(\mathsf{h}(x),\mathsf{a}) & \mathsf{f}(\mathsf{h}(x'),x) \to \mathsf{g}(x) & \mathsf{g}(x) \to \mathsf{f}(x,\mathsf{a}) \\ \mathsf{f}(x,x') \to \mathsf{f}(\mathsf{h}(x),\mathsf{a}) & \mathsf{f}(\mathsf{h}(x),x') \to \mathsf{g}(x) & \end{array}$$

Using the TRS $\mathcal{R}'_\#$ instead of $\mathcal{R}_\#$ we are now ready to extend Lemma 4.37 to right-linear TRSs. However, since we simulate non-left-linear rewrite rules by left-linear rewrite rules we can no longer guarantee that the set $\to^*_{\mathcal{R}'_\#}(L\sigma_\#)$ exactly represents the set $\mathsf{RFC}_L(\mathcal{R})\sigma_\#$.

Lemma 4.42. *Let $\mathcal{R}$ be a right-linear TRS and L a set of linear terms. We have* $\mathsf{RFC}_L(\mathcal{R})\sigma_\# \subseteq \to^*_{\mathcal{R}'_\#}(L\sigma_\#)$.

Proof. Applying Lemma 4.37 to the TRS $\mathsf{linear}(\mathcal{R})$ yields $\mathsf{RFC}_L(\mathsf{linear}(\mathcal{R}))\sigma_\# = \to^*_{\mathcal{R}'_\#}(L\sigma_\#)$. Hence it is sufficient to prove that the set $\mathsf{RFC}_L(\mathcal{R})\sigma_\#$ is a subset of $\mathsf{RFC}_L(\mathsf{linear}(\mathcal{R}))\sigma_\#$. First we show that every term $t \in \mathsf{RFC}_L(\mathcal{R})$ is linear. We use induction on the derivation of t. If $t \in \mathsf{rhs}(\mathcal{R})$ then t is linear because $\mathcal{R}$ is right-linear. Let $t = s[r]_p\sigma$ with $s \in \mathsf{RFC}_L(\mathcal{R})$, $l \to r$ a fresh variant of a rewrite rule in $\mathcal{R}$, and σ a most general unifier of $s|_p$ and l. According to the induction hypothesis s is linear. Hence $\mathcal{V}\mathsf{ar}(s|_p) \cap \mathcal{V}\mathsf{ar}(s[\Box]_p) = \varnothing$. From the linearity of r, $\mathcal{V}\mathsf{ar}(l) \cap \mathcal{V}\mathsf{ar}(s) = \varnothing$, $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(l)$, and the fact that σ is a most general unifier, we obtain that $r\sigma$ is linear and $\mathcal{V}\mathsf{ar}(r\sigma) \cap \mathcal{V}\mathsf{ar}(s\sigma[\Box]_p) = \varnothing$. It follows that t is linear.

Next we show that $\mathsf{RFC}_L(\mathcal{R}) \subseteq \mathsf{RFC}_L(\mathsf{linear}(\mathcal{R}))$, which immediately gives $\mathsf{RFC}_L(\mathcal{R})\sigma_\# \subseteq \mathsf{RFC}_L(\mathsf{linear}(\mathcal{R}))\sigma_\#$. Assume to the contrary that this does not hold. This is only possible if there are a term $t \in \mathsf{RFC}_L(\mathcal{R})$, a position $p \in \mathcal{P}\mathsf{os}_\mathcal{F}(t)$, a fresh variant $l \to r$ of a rewrite rule in $\mathcal{R}$, and a most general unifier σ of $t|_p$ and l such that $t[r]_p\sigma \in \mathsf{RFC}_L(\mathcal{R})$, and $t[r]_p\sigma \notin \mathsf{RFC}_L(\mathsf{linear}(\mathcal{R}))$. Since l and t do not share variables, we may assume without loss of generality that σ is idempotent. In order to arrive at a contradiction, we construct a term $l' \in \mathsf{linear}(l)$ and a most general unifier σ' of l' and $t|_p$ such that $t[r]_p\sigma' = t[r]_p\sigma$. Write $l = C[x_1,\ldots,x_n]$ with all variables displayed. Let $q_1,\ldots,q_n$ be the positions of these variables. Because σ is idempotent and t is linear, for every variable $x \in \mathcal{V}\mathsf{ar}(l)$ with $x\sigma \neq x$ there exists a position $q_x \in \{q_1,\ldots,q_n\}$ such that $x\sigma = t|_{pq_x}$. If $x\sigma = x$ we define $q_x = q_i$ for some $i \in \{1, \ldots, n\}$ such that $x_i = x$. Next we replace every variable x_i in l with $q_i \neq q_{x_i}$ and $i \in \{1, \ldots, n\}$ by a fresh variable. This yields a term $l' \in \mathsf{linear}(l)$. Let σ' be an idempotent most general unifier of l' and $t|_p$. It follows from the construction of l' that $\sigma(x) = \sigma'(x)$ for all variables $x \in \mathcal{V}\mathsf{ar}(l) \subseteq \mathcal{V}\mathsf{ar}(l')$. Since $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(l)$, $t[r]_p\sigma' = t[r]_p\sigma$ as desired. ☐

To guarantee the correctness of Lemma 4.42 it is important that $\mathsf{linear}(\mathcal{R})$ reflects all linearizations of the TRS $\mathcal{R}$. The following example shows what can go wrong if we would change the definition of $\mathsf{linear}(\mathcal{R})$ such that it represents an arbitrary linearization of $\mathcal{R}$.

Example 4.43. Consider the TRS $\mathcal{R}$ of Example 4.41. For the set $L = \mathsf{rhs}(\mathcal{R})$, $\mathsf{RFC}_L(\mathcal{R})\sigma_\#$ consists of the following terms:

$$\begin{array}{lll} \mathsf{g}(\mathsf{a}) & \mathsf{f}(\mathsf{a},\mathsf{a}) & \mathsf{f}(\mathsf{h}(\mathsf{a}),\mathsf{a}) \\ \mathsf{g}(\#) & \mathsf{f}(\#,\mathsf{a}) & \mathsf{f}(\mathsf{h}(\#),\mathsf{a}) \end{array}$$

If $\mathsf{linear}(\mathcal{R})$ would consist of the rewrite rules

$$\mathsf{f}(x,x') \to \mathsf{f}(\mathsf{h}(x),\mathsf{a}) \qquad \mathsf{f}(\mathsf{h}(x),x') \to \mathsf{g}(x) \qquad \mathsf{g}(x) \to \mathsf{f}(x,\mathsf{a})$$

then $\mathsf{RFC}_L(\mathsf{linear}(\mathcal{R}))\sigma_\# = \{\mathsf{g}(\mathsf{h}^i(\#)), \mathsf{f}(\mathsf{h}^i(\#),\mathsf{a}) \mid i \geqslant 0\}$. Note that $\mathsf{g}(\mathsf{a})$ is missing, invalidating Lemma 4.42. In the proof the linearization $\mathsf{f}(\mathsf{h}(y'),y) \to \mathsf{g}(y)$ of the variant $\mathsf{f}(\mathsf{h}(y),y) \to \mathsf{g}(y)$ is constructed because the right-hand side $\mathsf{f}(\mathsf{h}(x),\mathsf{a})$ is unified with $\mathsf{f}(\mathsf{h}(y),y)$ to produce the term $\mathsf{g}(\mathsf{a})$ and only the second occurrence of y is mapped to the subterm a of $\mathsf{f}(\mathsf{h}(y),\mathsf{a})$. We remark that the TRS $\mathcal{R}$ is non-terminating since it admits the cycle rewrite sequence $\mathsf{g}(\mathsf{a}) \to_\mathcal{R} \mathsf{f}(\mathsf{a},\mathsf{a}) \to_\mathcal{R} \mathsf{f}(\mathsf{h}(\mathsf{a}),\mathsf{a}) \to_\mathcal{R} \mathsf{g}(\mathsf{a})$. However, it is easy to see that $\mathcal{R}$ is terminating on $\mathsf{RFC}_L(\mathsf{linear}(\mathcal{R}))\sigma_\#$: $\mathsf{f}(\mathsf{h}^i(\#),\mathsf{a})$ is a normal form and $\mathsf{g}(\mathsf{h}^i(\#))$ rewrites only to $\mathsf{f}(\mathsf{h}^i(\#),\mathsf{a})$, for all $i \geqslant 0$.

The following example shows that the reverse inclusion of Lemma 4.42 does not hold.

Example 4.44. For the TRS $\mathcal{R}$ of Example 4.41 the set $\to^*_{\mathcal{R}'_\#}(L\sigma_\#)$, where $L = \mathsf{rhs}(\mathcal{R})$, consists of all terms

$$\mathsf{g}(\mathsf{h}^i(\#)) \qquad \mathsf{g}(\mathsf{h}^i(\mathsf{a})) \qquad \mathsf{f}(\mathsf{h}^i(\#),\mathsf{a}) \qquad \mathsf{f}(\mathsf{h}^i(\mathsf{a}),\mathsf{a})$$

with $i \geqslant 0$. Because $\mathsf{RFC}_L(\mathcal{R})\sigma_\#$ is a finite set consisting of the terms $\mathsf{g}(\mathsf{a})$, $\mathsf{f}(\mathsf{a},\mathsf{a})$, $\mathsf{f}(\mathsf{h}(\mathsf{a}),\mathsf{a})$, $\mathsf{g}(\#)$, $\mathsf{f}(\#,\mathsf{a})$, and $\mathsf{f}(\mathsf{h}(\#),\mathsf{a})$ it is obvious that $\mathsf{RFC}_L(\mathcal{R})\sigma_\#$ is a strict subset of $\to^*_{\mathcal{R}'_\#}(L\sigma_\#)$.

Theorem 4.45. *Let $\mathcal{R}$ be a right-linear TRS. If $\mathcal{R}$ is match-raise-bounded for $\to^*_{\mathcal{R}'_\#}(\mathsf{rhs}(\mathcal{R})\sigma_\#)$ then $\mathcal{R}$ is terminating.*

Proof. Since the set $\mathsf{RFC}_{\mathsf{rhs}(\mathcal{R})}(\mathcal{R})\sigma_\#$ is a subset of $\to^*_{\mathcal{R}'_\#}(\mathsf{rhs}(\mathcal{R})\sigma_\#)$, according to Lemma 4.42, $\mathcal{R}$ is also match-raise-bounded for $\mathsf{RFC}_{\mathsf{rhs}(\mathcal{R})}(\mathcal{R})\sigma_\#$. Theorem 4.15 yields the termination of $\mathcal{R}$ on $\mathsf{RFC}_{\mathsf{rhs}(\mathcal{R})}(\mathcal{R})\sigma_\#$. Since rewriting is closed under substitutions, $\mathcal{R}$ is terminating on $\mathsf{RFC}_{\mathsf{rhs}(\mathcal{R})}(\mathcal{R})$. From Theorem 4.35 we conclude that $\mathcal{R}$ is terminating on all terms. ☐

In order to show that a TRS $\mathcal{R}$ is match(-raise)-bounded for the language $L = \to^*_{\mathcal{R}_\#}(\mathsf{rhs}(\mathcal{R})\sigma_\#)$ ($L = \to^*_{\mathcal{R}'_\#}(\mathsf{rhs}(\mathcal{R})\sigma_\#)$) we have to construct a (quasi-deterministic and raise-consistent) tree automaton that is (quasi-)compatible with $\mathsf{match}(\mathcal{R})$ and $\mathsf{lift}_0(L)$. We do that by performing two steps. First we construct a tree automaton $\mathcal{A}$ that is compatible with $\mathcal{R}_\#$ ($\mathcal{R}'_\#$) and $\mathsf{rhs}(\mathcal{R})\sigma_\#$. Since $\mathcal{R}_\#$ ($\mathcal{R}'_\#$) is left-linear we know by Theorem 3.3 that $\mathcal{L}(\mathcal{A}) \supseteq L$. In a second step we search for a (quasi-deterministic and raise-consistent) tree automaton that is (quasi-)compatible with $\mathsf{match}(\mathcal{R})$ and $\mathsf{lift}_0(\mathcal{L}(\mathcal{A}))$ as described in Section 4.4. If such an automaton has been found we know that the TRS $\mathcal{R}$ is match(-raise)-bounded for L. Note that if $\mathcal{R}$ is left-linear the two steps can be combined in an optimized way [24]. Instead of computing an intermediate tree automaton that accepts all terms in $\to^*_{\mathcal{R}_\#}(\mathsf{rhs}(\mathcal{R})\sigma_\#)$, we immediately start with the construction of a tree automaton that is (quasi-)compatible with the TRS $\mathsf{match}(\mathcal{R}) \cup \mathsf{lift}_0(\mathcal{R}_\# \setminus \mathcal{R})$ and $\mathsf{lift}_0(\mathsf{rhs}(\mathcal{R})\sigma_\#)$. It is not difficult to observe that $\mathcal{R}$ is match-bounded for $\to^*_{\mathcal{R}_\#}(\mathsf{rhs}(\mathcal{R})\sigma_\#)$ whenever the construction terminates.

Example 4.46. Let $\mathcal{R}$ be the TRS of Example 4.2. We show that $\mathcal{R}$ is match-bounded for the language $\to^*_{\mathcal{R}_\#}(\mathsf{rhs}(\mathcal{R})\sigma_\#)$ by 0. To compute the set $\to^*_{\mathcal{R}_\#}(\mathsf{rhs}(\mathcal{R})\sigma_\#)$ we close the initial tree automaton $\mathcal{A}$ consisting of the final state 4 and the transitions

$$\# \to 1 \qquad \mathsf{f}(1) \to 2 \qquad \mathsf{h}(2) \to 3 \qquad \mathsf{g}(3,1) \to 4$$

under rewriting. Note that $\mathcal{A}$ accepts the language $\mathsf{rhs}(\mathcal{R})\sigma_\#$. Since $\mathsf{f}(\#) \to_{\mathcal{R}_\#} \mathsf{g}(\mathsf{h}(\mathsf{f}(\#)),\#)$ and $\mathsf{f}(\#) \to^* 2$ we have to establish $\mathsf{g}(\mathsf{h}(\mathsf{f}(\#)),\#) \to^* 2$. We do that by reusing the transitions $\mathsf{f}(1) \to 2$ and $\mathsf{h}(2) \to 3$ and by adding the new transition $\mathsf{g}(3,1) \to 2$. After that step, the obtained tree automaton is already compatible with $\mathcal{R}_\#$ and $\mathsf{rhs}(\mathcal{R})\sigma_\#$. Hence $\mathcal{L}(\mathcal{A}) \supseteq \to^*_{\mathcal{R}_\#}(\mathsf{rhs}(\mathcal{R})\sigma_\#)$. Next we transform $\mathcal{A}$ into a tree automaton over the signature $\{\#_0, \mathsf{f}_0, \mathsf{g}_0, \mathsf{h}_0\}$ by labeling its function symbols with height 0. So we obtain the following tree automaton:

$$\#_0 \to 1 \qquad \mathsf{f}_0(1) \to 2 \qquad \mathsf{h}_0(2) \to 3 \qquad \mathsf{g}_0(3,1) \to 2 \mid 4$$

After that it remains to close this tree automaton under rewriting with respect to $\mathsf{match}(\mathcal{R})$. It is not difficult to see that the current automaton is already compatible with $\mathsf{match}(\mathcal{R})$. Hence $\mathcal{R}$ is match-bounded for $\to^*_{\mathcal{R}_\#}(\mathsf{rhs}(\mathcal{R})\sigma_\#)$ by 0 and therefore terminating.

4.6 Summary

In this chapter we extended the match-bound technique by removing the left-linearity restriction. First we introduced so called *raise-rules* to ensure that the theory on which the method is based works for non-left-linear TRSs. After that we presented how *tree automata completion* can be used to automatically obtain certificates for the e(-raise)-boundedness of TRSs. To ensure that for a non-left-linear TRS the obtained tree automaton is closed under the implicit raise-steps caused by the rewrite relation $\xrightarrow{\mathsf{r}}$ we introduced *raise-consistent* tree automata. Finally we showed how to strengthen the method by taking forward closures into account.

Chapter 5

The Dependency Pair Framework

The *dependency pair method*, developed by Arts and Giesl [1], is a powerful approach for proving the termination of rewrite systems. In difference to other methods it does not prove the termination of a rewrite system directly. Instead it transforms the given rewrite system into a set of ordering constraints by analyzing the recursive calls in the original system. Afterwards the ordering constraints are solved by applying some standard termination techniques. Initially proposed as an additional method to prove the termination of rewrite systems, Giesl, Thiemann, and Schneider-Kamp showed in [27], that dependency pairs can be used as a general concept to integrate and combine several termination techniques for termination analysis. This modular reformulation and improvement of the dependency pair method result in the well-known *dependency pair framework* [30, 55]. On the basis of its ability to combine different termination techniques, the dependency pair framework became the most popular technique to prove the termination of rewrite systems automatically.

After presenting a simplified version of the dependency pair framework which is sufficient for our purposes, we show in Section 5.2 how the match-bound technique can be successfully integrated into this framework. The key to this extension is the introduction of two new enrichments which exploit the special properties of dependency pair problems. Afterwards, in Section 5.3 we focus on the *dependency graph processor*, one of the most frequently used techniques within the dependency pair framework. We show that by using tree automata completion, efficient and powerful approximations of the dependency graph can be obtained which sometimes even allow us to eliminate arcs from the real dependency graph. Finally, in Section 5.4 we increase the power of the techniques introduced in the previous sections by combining them with *usable rules.*

Many of the results presented in this chapter appeared already in the conference papers [41, 42] as well as the journal paper [43]. New contributions include an optimized definition of c-innermost dependency graphs as well as c-improved innermost dependency graphs presented in Subsection 5.3.5. Furthermore, in Subsection 5.4.2 we analyze and explain in detail how the (innermost) dependency graph approximations as well as the improved (innermost) dependency graph approximations based one tree automata completion can be combined with usable rules.

5.1 Preliminaries

A *directed graph* $\mathcal{G}$ is a pair (N, E) where N is a finite set of nodes and $E \subseteq N \times N$ a finite set of arcs. Here an arc $(\alpha, \beta) \in E$ is considered to be directed from α to β. Let $\mathcal{G} = (N, E)$ and $\mathcal{G}' = (N', E')$ be two directed graphs. The intersection of $\mathcal{G}$ and $\mathcal{G}'$ is defined as $\mathcal{G} \cap \mathcal{G}' = (N \cap N', E \cap E')$. Let $M \subseteq N$ be a set of nodes. The subgraph of $\mathcal{G}$ induced by removing the nodes in M from N is denoted by $\mathcal{G} \setminus M$ and defined as $\mathcal{G} \setminus M = (N', E \cap (N' \times N'))$ where $N' = N \setminus M$. We often write $(\alpha, \beta) \in \mathcal{G}$ to denote that $(\alpha, \beta) \in E$.

Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$. The signature $\mathcal{F}$ is extended with symbols $f^\#$ for every defined symbol $f \in \mathcal{F}\mathsf{un}_\mathcal{D}(\mathcal{R})$, where $f^\#$ has the same arity as f, resulting in the signature $\mathcal{F}^\#$. If $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ with $\mathsf{root}(t) \in \mathcal{F}\mathsf{un}_\mathcal{D}(\mathcal{R})$ then $t^\#$ denotes the term that is obtained from t by replacing its root symbol with $\mathsf{root}(t)^\#$. If $l \to r \in \mathcal{R}$ and t is a subterm of r with a defined root symbol that is not a proper subterm of l then the rule $l^\# \to t^\#$ is a *dependency pair* of $\mathcal{R}$. The set of dependency pairs of $\mathcal{R}$ is denoted by $\mathsf{DP}(\mathcal{R})$. Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. A *dependency pair problem* (DP problem for short) is a triple $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ such that the root symbols of the left- and right-hand sides of $\mathcal{P}$ do neither occur in $\mathcal{R}$ nor in proper subterms of the left and right-hand sides of rules in $\mathcal{P}$ and $\mathcal{G} = (\mathcal{P}, E)$ is a directed graph.[1] A DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is said to be linear (left-linear, right-linear, duplicating) if $\mathcal{P} \cup \mathcal{R}$ is linear (left-linear, right-linear, duplicating respectively). A DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is called *finite* if there are no infinite rewrite sequences of the form $s_1 \xrightarrow{\epsilon}_{\alpha_1} t_1 \to^*_\mathcal{R} s_2 \xrightarrow{\epsilon}_{\alpha_2} t_2 \to^*_\mathcal{R} \cdots$ such that all terms t_1, t_2, $\ldots$ are terminating with respect to $\mathcal{R}$ and $(\alpha_i, \alpha_{i+1}) \in \mathcal{G}$ for all $i \geqslant 1$. Such an infinite sequence is said to be *minimal*. Here the ϵ in $\xrightarrow{\epsilon}_\mathcal{P}$ denotes that the application of the rule in $\mathcal{P}$ takes place at the root position. We say that $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite *on* a language $L \subseteq \mathcal{T}(\mathcal{F}^\#)$ if there is no minimal rewrite sequence starting at a term s in L. Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a DP problem and $\mathcal{S} \subseteq \mathcal{P}$ a set of rewrite rules. We write $(\mathcal{P}, \mathcal{R}, \mathcal{G}) \setminus \mathcal{S}$ to denote the DP problem obtained from $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ by removing the rules in $\mathcal{S}$ from $\mathcal{P}$, that is, $(\mathcal{P}, \mathcal{R}, \mathcal{G}) \setminus \mathcal{S} = (\mathcal{P} \setminus \mathcal{S}, \mathcal{R}, \mathcal{G} \setminus \mathcal{S})$.

Example 5.1. Consider the TRS $\mathcal{R}$ over the signature $\mathcal{F} = \{\mathsf{a}, \mathsf{f}, \mathsf{g}, \mathsf{h}\}$ consisting of the following two rewrite rules:

$$\mathsf{f}(\mathsf{g}(x), y) \to \mathsf{g}(\mathsf{h}(x, y)) \qquad\qquad \mathsf{h}(x, y) \to \mathsf{f}(x, \mathsf{g}(y))$$

The dependency pairs of $\mathcal{R}$ over the signature $\mathcal{F}^\# = \{\mathsf{a}, \mathsf{f}, \mathsf{g}, \mathsf{h}, \mathsf{f}^\#, \mathsf{h}^\#\}$ are $\mathsf{f}^\#(\mathsf{g}(x), y) \to \mathsf{h}^\#(x, y)$ and $\mathsf{h}^\#(x, y) \to \mathsf{f}^\#(x, \mathsf{g}(y))$. The initial DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ consistent with $\mathcal{R}$ is defined as $\mathcal{P} = \mathsf{DP}(\mathcal{R})$ and $\mathcal{G} = (\mathcal{P}, \mathcal{P} \times \mathcal{P})$ where $\mathcal{G}$ consists of 4 arcs. In the following we often write F instead of $\mathsf{f}^\#$, etc. to ease readability.

The main result underlying the dependency pair framework states that a TRS $\mathcal{R}$ is terminating if and only if the initial DP problem $(\mathsf{DP}(\mathcal{R}), \mathcal{R}, \mathcal{G})$

[1]We remark that in Section 5.2 the third component of a DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$—namely the graph $\mathcal{G}$—does not play a role since we operate just on the TRSs $\mathcal{P}$ and $\mathcal{R}$. In Section 5.3 this will then change because we manipulate $\mathcal{G}$ to prove finiteness of DP problems.

with $\mathcal{G} = (\mathsf{DP}(\mathcal{R}), \mathsf{DP}(\mathcal{R}) \times \mathsf{DP}(\mathcal{R}))$ is finite. In order to prove finiteness of a DP problem a number of so called *dependency pair processors* (DP processors for short) have been developed. DP processors are functions that take a DP problem as input and return a set of DP problems as output. In order to be employed to prove termination they need to be *sound*, that is, if all DP problems in a set returned by a DP processor are finite then the initial DP problem is finite. In addition, to ensure that a DP processor can be used to prove non-termination it must be *complete* which means that if one of the DP problems returned by the DP processor is not finite then the original DP problem is not finite. The general procedure for proving finiteness of a DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ tries to remove step by step those rewrite rules in $\mathcal{P}$ which cannot be used infinitely often in any minimal rewrite sequence. In each step a different DP processor can be applied. As soon as $\mathcal{P}$ is empty or a empty list of DP problems is returned, we can conclude that the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite.

5.2 Combining Dependency Pairs and Bounds

In this section we incorporate the match-bound technique into the dependency pair framework. To guarantee a successful integration we need to modularize the method in order to be able to simplify DP problems. We achieve this by introducing two new enrichments which exploit the special properties of DP problems. To simplify the presentation we first consider left-linear TRSs. The extension to non-left-linear TRSs is discussed in Subsection 5.2.2. Finally in Subsection 5.2.3 it is explained how (quasi-deterministic and raise-consistent) tree automata can be used to infer finiteness of DP problems.

5.2.1 DP-Bounds for Left-Linear DP Problems

It is easy to incorporate the match-bound technique into the dependency pair framework by defining a processor that checks for e-boundedness of $\mathcal{P} \cup \mathcal{R}$.

Theorem 5.2. *The DP processor*

$$(\mathcal{P}, \mathcal{R}, \mathcal{G}) \mapsto \begin{cases} \varnothing & \text{\textit{if} } \mathcal{P} \cup \mathcal{R} \text{ \textit{is left-linear and either top-bounded}} \\ & \text{\textit{or roof-bounded, or linear and match-bounded}} \\ & \text{\textit{for} } \mathcal{T}(\mathcal{F}) \\ \{(\mathcal{P}, \mathcal{R}, \mathcal{G})\} & \text{\textit{otherwise}} \end{cases}$$

where $\mathcal{F}$ is the signature of $\mathcal{P} \cup \mathcal{R}$, is sound and complete.

Proof. Assume that $\mathcal{P} \cup \mathcal{R}$ is e-bounded for $\mathcal{T}(\mathcal{F})$ with $e \in \{\mathsf{top}, \mathsf{roof}, \mathsf{match}\}$. By Theorem 4.4 we conclude that $\mathcal{P} \cup \mathcal{R}$ is terminating. Because $\mathcal{P} \cup \mathcal{R}$ does not admit an infinite rewrite sequence we know that $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ does not admit a minimal rewrite sequence. Hence $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite. □

This DP processor either succeeds by proving that the combined TRS $\mathcal{P} \cup \mathcal{R}$ is e-bounded or, when the e-boundedness of $\mathcal{P} \cup \mathcal{R}$ cannot be proved, it returns the initial DP problem. Since the construction of a compatible tree automaton does not terminate for TRSs that are not e-bounded, the latter situation typically does not happen. Hence the DP processor of Theorem 5.2 is applicable only at the leaves of the dependency pair search tree, which means that it cannot be used to (partly) simplify a DP problem. Below we address this problem by adapting the match-bound technique in such a way that it can remove single rules of $\mathcal{P}$. We introduce two new enrichments, namely $\mathsf{top\text{-}DP}(\mathcal{P}, s \to t, \mathcal{R})$ and $\mathsf{match\text{-}DP}(\mathcal{P}, s \to t, \mathcal{R})$ to achieve this. The basic idea behind these TRSs is that every minimal sequence of $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ in which $s \to t$, the rule that is to be removed from $\mathcal{P}$, is used infinitely often is simulated by a height increasing infinite rewrite sequence. To simplify the presentation in the remaining part we abbreviate the TRS $(\mathcal{P} \setminus \{s \to t\}) \cup \mathcal{R}$ by $\mathcal{S}_{s \to t}$.

Definition 5.3. Let $\mathcal{S}$ be a TRS over a signature $\mathcal{F}$. The TRS $e\text{-}\mathsf{DP}(\mathcal{S})$ over the signature $\mathcal{F}_{\mathbb{N}}$ consists of all rules $l' \to \mathsf{lift}_c(r)$ such that

$$c = \min\left(\{\mathsf{height}(l'(\epsilon))\} \cup \{1 + \mathsf{height}(l'(p)) \mid p \in e(\mathsf{base}(l'), r)\}\right)$$

and $\mathsf{base}(l') \to r \in \mathcal{S}$. Given a DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ and a rule $s \to t \in \mathcal{P}$, the TRS $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$ is defined as the union of $e\text{-}\mathsf{DP}(\mathcal{S}_{s \to t})$ and $e(s \to t)$. Let $c \in \mathbb{N}$. The restriction of $e\text{-}\mathsf{DP}(\mathcal{S})$ and $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$ to the signature $\mathcal{F}_{\{0,\dots,c\}}$ is denoted by $e\text{-}\mathsf{DP}_c(\mathcal{S})$ and $e\text{-}\mathsf{DP}_c(\mathcal{P}, s \to t, \mathcal{R})$.

Example 5.4. Consider the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ consisting of the rewrite rules $\mathsf{f}(\mathsf{g}(x), y) \to \mathsf{g}(\mathsf{h}(x, y))$ and $\mathsf{h}(x, y) \to \mathsf{f}(x, \mathsf{g}(y))$, $\mathcal{P} = \mathsf{DP}(\mathcal{R})$ consisting of $\mathsf{F}(\mathsf{g}(x), y) \to \mathsf{H}(x, y)$ and $\mathsf{H}(x, y) \to \mathsf{F}(x, \mathsf{g}(y))$, and $\mathcal{G} = (\mathcal{P}, \mathcal{P} \times \mathcal{P})$. Let $s \to t$ be the first of the two dependency pairs. Then $\mathsf{match\text{-}DP}(\mathcal{R})$ contains the rules

$$\begin{array}{ll}
\mathsf{f}_0(\mathsf{g}_0(x), y) \to \mathsf{g}_0(\mathsf{h}_0(x, y)) & \mathsf{h}_0(x, y) \to \mathsf{f}_0(x, \mathsf{g}_0(y)) \\
\mathsf{f}_0(\mathsf{g}_1(x), y) \to \mathsf{g}_0(\mathsf{h}_0(x, y)) & \mathsf{h}_1(x, y) \to \mathsf{f}_1(x, \mathsf{g}_1(y)) \\
\mathsf{f}_2(\mathsf{g}_0(x), y) \to \mathsf{g}_1(\mathsf{h}_1(x, y)) & \cdots
\end{array}$$

$\mathsf{match\text{-}DP}(\mathcal{P} \setminus \{s \to t\})$ contains

$$\begin{array}{ll}
\mathsf{H}_0(x, y) \to \mathsf{F}_0(x, \mathsf{g}_0(y)) & \mathsf{H}_1(x, y) \to \mathsf{F}_1(x, \mathsf{g}_1(y)) \\
\mathsf{H}_2(x, y) \to \mathsf{F}_2(x, \mathsf{g}_2(y)) & \cdots
\end{array}$$

and the rewrite rules

$$\begin{array}{ll}
\mathsf{F}_0(\mathsf{g}_0(x), y) \to \mathsf{H}_1(x, y) & \mathsf{F}_1(\mathsf{g}_0(x), y) \to \mathsf{H}_1(x, y) \\
\mathsf{F}_0(\mathsf{g}_1(x), y) \to \mathsf{H}_1(x, y) & \cdots
\end{array}$$

belong to $\mathsf{match}(s \to t)$. The union of these three TRSs constitutes the TRS $\mathsf{match\text{-}DP}(\mathcal{P}, s \to t, \mathcal{R})$. If we replace $\mathsf{match}(s \to t)$ by $\mathsf{match\text{-}DP}(\{s \to t\})$, which consists of the rules

$$\begin{array}{ll}
\mathsf{F}_0(\mathsf{g}_0(x), y) \to \mathsf{H}_0(x, y) & \mathsf{F}_1(\mathsf{g}_0(x), y) \to \mathsf{H}_1(x, y) \\
\mathsf{F}_0(\mathsf{g}_1(x), y) \to \mathsf{H}_0(x, y) & \cdots
\end{array}$$

we obtain the TRS $\mathsf{match\text{-}DP}(\mathcal{P} \cup \mathcal{R})$. Note that all TRSs have infinitely many rewrite rules.

The idea now is to use the enrichment $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$ to simplify the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ into $(\mathcal{P}, \mathcal{R}, \mathcal{G}) \setminus \{s \to t\}$. For that we need the property defined below.

Definition 5.5. Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a DP problem and $s \to t \in \mathcal{P}$ a rewrite rule. We call $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ *e-DP-bounded* for $s \to t$ and a language L if there exists a number $c \in \mathbb{N}$ such that the height of function symbols occurring in terms in $\to^*_{e\text{-}\mathsf{DP}(\mathcal{P},s\to t,\mathcal{R})}(\mathsf{lift}_0(L))$ is at most c.

To ensure that the TRS $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$ can assist to prove finiteness of the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$, it is crucial that every minimal rewrite sequence in $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ with infinitely many $\xrightarrow{\epsilon}_{s\to t}$-steps can be simulated by an infinite height increasing sequence in $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$. To this end it is important that rewrite rules in $e\text{-}\mathsf{DP}(\mathcal{S}_{s\to t})$ do not propagate the minimal height of the contracted redex unless the height of the root symbol of the redex is minimal. This is the reason for the slightly complicated definition of c in Definition 5.3. The following example shows what goes wrong if we would simplify the definition.

Example 5.6. Consider the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ with $\mathcal{R}$ consisting of the rewrite rules $\mathsf{f}(x) \to \mathsf{g}(x)$ and $\mathsf{g}(\mathsf{h}(x)) \to \mathsf{f}(\mathsf{h}(x))$, $\mathcal{P}$ consisting of the dependency pairs $\mathsf{F}(x) \to \mathsf{G}(x)$ and $\mathsf{G}(\mathsf{h}(x)) \to \mathsf{F}(\mathsf{h}(x))$ of $\mathcal{R}$, and $\mathcal{G} = (\mathcal{P}, \mathcal{P} \times \mathcal{P})$. The DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is not finite because the term $\mathsf{F}(\mathsf{h}(x))$ admits a minimal rewrite sequence. If we change the definition of c in Definition 5.3 to

$$c = \min\{\mathsf{height}(l'(p)) \mid p \in e(\mathsf{base}(l'), r)\}$$

then for $s \to t = \mathsf{F}(x) \to \mathsf{G}(y)$ we have

$$\mathsf{F}_0(\mathsf{h}_0(x)) \xrightarrow{\epsilon}_{\mathsf{match}(s\to t)} \mathsf{G}_1(\mathsf{h}_0(x)) \xrightarrow{\epsilon}_{\mathsf{match}\text{-}\mathsf{DP}(\mathcal{P}\setminus\{s\to t\})} \mathsf{F}_0(\mathsf{h}_0(x))$$

and it would follow that $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is match-DP-bounded for $\mathsf{F}(x) \to \mathsf{G}(x)$. As we will see later, this would imply that we can remove $\mathsf{F}(x) \to \mathsf{G}(x)$ from $\mathcal{P}$. Because the remaining DP problem is finite we would falsely conclude the termination of the original TRS $\mathcal{R}$.

An immediate consequence of the next lemma is that every derivation caused by some DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ can be simulated using the rewrite rules in $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$.

Lemma 5.7. *Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a left-linear DP problem and $s \to t \in \mathcal{P}$. If $u \to_{s\to t} v$ or $u \to_{\mathcal{S}_{s\to t}} v$ then for all terms u' with $\mathsf{base}(u') = u$ there exists a term v' such that $\mathsf{base}(v') = v$ and $u' \to_{e(s\to t)} v'$ or $u' \to_{e\text{-}\mathsf{DP}(\mathcal{S}_{s\to t})} v'$.*

Proof. Straightforward. □

To be able to use the concept of e-DP-boundedness to simplify DP problems, we need to ensure that no restriction of $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$ to a finite signature admits minimal rewrite sequences with infinitely many $\xrightarrow{\epsilon}_{e(s\to t)}$-steps. For the TRS $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$ with $e = \mathsf{top}$ this is shown below. Note that if we use $e\text{-}\mathsf{DP}(\mathcal{P} \cup \mathcal{R})$ instead of $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$ then this property does not hold because every rewrite sequence in $\mathcal{P} \cup \mathcal{R}$ can be simulated by an $e\text{-}\mathsf{DP}_0(\mathcal{P} \cup \mathcal{R})$ sequence.

Lemma 5.8. *Let $(\mathcal{P},\mathcal{R},\mathcal{G})$ be a DP problem, $s \to t \in \mathcal{P}$ a rewrite rule, and $c \geqslant 0$. The TRS $\mathsf{top\text{-}DP}_c(\mathcal{P}, s \to t, \mathcal{R})$ does not admit rewrite sequences with infinitely many $\xrightarrow{\epsilon}_{\mathsf{top}(s\to t)}$-steps.*

Proof. Assume to the contrary that there is such an infinite rewrite sequence

$$s_1 \xrightarrow{\epsilon}_{\mathsf{top}_c(s\to t)} t_1 \to^*_{\mathsf{top\text{-}DP}_c(\mathcal{S}_{s\to t})} s_2 \xrightarrow{\epsilon}_{\mathsf{top}_c(s\to t)} t_2 \to^*_{\mathsf{top\text{-}DP}_c(\mathcal{S}_{s\to t})} \cdots$$

emanating from a term s_1. Because the root symbols in $\mathcal{P}$ do not appear anywhere else in $\mathcal{P}$ or $\mathcal{R}$, we know that only rewrite rules from $\mathsf{top}(s \to t)$ and $\mathsf{top\text{-}DP}(\mathcal{P} \setminus \{s \to t\})$ are applied at root positions. Every rewrite rule $l \to r$ in $\mathsf{top\text{-}DP}(\mathcal{P} \setminus \{s \to t\})$ has the property that $\mathsf{height}(l(\epsilon))$ is equivalent to $\mathsf{height}(r(\epsilon))$. Hence $\mathsf{height}(t_i(\epsilon)) = \mathsf{height}(s_{i+1}(\epsilon))$ for all $i \geqslant 1$. By definition, for every $l \to r \in \mathsf{top}(s \to t)$ we have $\mathsf{height}(r(\epsilon)) = \mathsf{height}(l(\epsilon)) + 1$ and thus $\mathsf{height}(t_i(\epsilon)) = \mathsf{height}(s_i(\epsilon)) + 1$ for all $i \geqslant 1$. It follows that $\mathsf{height}(t_{c+1}(\epsilon)) \geqslant c+1$, contradicting the assumption. □

Theorem 5.9. *Let $(\mathcal{P},\mathcal{R},\mathcal{G})$ be a DP problem, $s \to t \in \mathcal{P}$ a rewrite rule, and L a language such that $(\mathcal{P},\mathcal{R},\mathcal{G})$ is top-DP-bounded for $s \to t$ and L. If $\mathcal{P} \cup \mathcal{R}$ is left-linear then $(\mathcal{P},\mathcal{R},\mathcal{G})$ is finite on L if and only if $(\mathcal{P},\mathcal{R},\mathcal{G}) \setminus \{s \to t\}$ is finite on L.*

Proof. The only-if direction is trivial. For the if direction, suppose that the DP problem $(\mathcal{P},\mathcal{R},\mathcal{G}) \setminus \{s \to t\}$ is finite on L. If $(\mathcal{P},\mathcal{R},\mathcal{G})$ is not finite on L then there exists a minimal rewrite sequence

$$s_1 \xrightarrow{\epsilon}_{s\to t} t_1 \to^*_{\mathcal{S}_{s\to t}} s_2 \xrightarrow{\epsilon}_{s\to t} t_2 \to^*_{\mathcal{S}_{s\to t}} s_3 \xrightarrow{\epsilon}_{s\to t} \cdots$$

with $s_1 \in L$. Due to left-linearity, this sequence can be lifted to an infinite $\mathsf{top\text{-}DP}(\mathcal{P}, s \to t, \mathcal{R})$ rewrite sequence starting from $\mathsf{lift}_0(s_1)$ using Lemma 5.7. Since the original sequence contains infinitely many $\xrightarrow{\epsilon}_{s\to t}$-steps the lifted sequence contains infinitely many $\xrightarrow{\epsilon}_{\mathsf{top}(s\to t)}$-steps. Moreover, because $(\mathcal{P},\mathcal{R},\mathcal{G})$ is top-DP-bounded for L, there is a $c \geqslant 0$ such that the height of every function symbol occurring in a term in the lifted sequence is at most c. Hence the employed rewrite rules must come from $\mathsf{top\text{-}DP}_c(\mathcal{P}, s \to t, \mathcal{R})$ and therefore $\mathsf{top\text{-}DP}_c(\mathcal{P}, s \to t, \mathcal{R})$ contains a minimal rewrite sequence consisting of infinitely many $\xrightarrow{\epsilon}_{\mathsf{top}(s\to t)}$-steps. This however is excluded by Lemma 5.8. □

If we restrict Lemma 5.8 to *minimal* rewrite sequences, it also holds for $e = \mathsf{match}$ provided that the TRSs $\mathcal{R}$ and $\mathcal{P}$ are non-duplicating. The proof is considerably more complicated and given in Appendix B.2.

Lemma 5.10. *Let $(\mathcal{P},\mathcal{R},\mathcal{G})$ be a DP problem, $s \to t \in \mathcal{P}$ a rewrite rule, and $c \geqslant 0$. If $\mathcal{P} \cup \mathcal{R}$ is non-duplicating then the TRS $\mathsf{match\text{-}DP}_c(\mathcal{P}, s \to t, \mathcal{R})$ does not admit minimal rewrite sequences with infinitely many $\xrightarrow{\epsilon}_{\mathsf{match}(s\to t)}$-steps.*

Theorem 5.11. *Let $(\mathcal{P},\mathcal{R},\mathcal{G})$ be a DP problem, $s \to t \in \mathcal{P}$ a rewrite rule, and L a language such that $(\mathcal{P},\mathcal{R},\mathcal{G})$ is match-DP-bounded for $s \to t$ and L. If $\mathcal{P} \cup \mathcal{R}$ is linear then $(\mathcal{P},\mathcal{R},\mathcal{G})$ is finite on L if and only if $(\mathcal{P},\mathcal{R},\mathcal{G}) \setminus \{s \to t\}$ is finite on L.*

Proof. Similarly to the proof of Theorem 5.9, using Lemma 5.10 instead of Lemma 5.8. Note that in the presence of left-linearity, the non-duplicating requirement in Lemma 5.10 is equivalent to linearity. □

We conjecture that Lemma 5.8 also holds for $e = \mathsf{roof}$. A positive solution is important as roof-bounds are strictly more powerful than top-bounds (see Section 7.1 and [24]).

Theorem 5.12. *The DP processor*

$$(\mathcal{P},\mathcal{R},\mathcal{G}) \mapsto \begin{cases} \{(\mathcal{P},\mathcal{R},\mathcal{G}) \setminus \{s \to t\}\} & \text{if } (\mathcal{P},\mathcal{R},\mathcal{G}) \text{ is top-DP-bounded and} \\ & \text{left-linear or match-DP-bounded} \\ & \text{and linear for } s \to t \text{ and } \mathcal{T}(\mathcal{F}) \\ \{(\mathcal{P},\mathcal{R},\mathcal{G})\} & \text{otherwise} \end{cases}$$

where $\mathcal{F}$ is the signature of $\mathcal{P} \cup \mathcal{R}$, is sound and complete.

Proof. Immediate consequence of Theorems 5.9 and 5.11. □

Example 5.13. The DP problem $(\mathcal{P},\mathcal{R},\mathcal{G})$ of Example 5.4 over the signature $\mathcal{F} = \{\mathsf{a},\mathsf{f},\mathsf{g},\mathsf{h},\mathsf{F},\mathsf{H}\}$ is match-DP-bounded for $\mathsf{F}(\mathsf{g}(x),y) \to \mathsf{H}(x,y)$ and $\mathcal{T}(\mathcal{F})$ by 1. So by applying the above DP processor we obtain the new DP problem $(\mathcal{P}',\mathcal{R},\mathcal{G}')$ with $\mathcal{P}' = \{\mathsf{H}(x,y) \to \mathsf{F}(x,\mathsf{g}(y))\}$ and $\mathcal{G}' = \mathcal{G} \setminus (\mathcal{P} \setminus \mathcal{P}')$. In Subsection 5.2.3 it is explained in detail how e-DP-boundedness can be automatically checked.

5.2.2 Raise-DP-Bounds for Non-Left-Linear DP Problems

In order to apply the DP processor of Theorem 5.2 to non-left-linear TRSs, we use e-raise-bounds instead of e-bounds.

Theorem 5.14. *The DP processor*

$$(\mathcal{P},\mathcal{R},\mathcal{G}) \mapsto \begin{cases} \varnothing & \text{if } \mathcal{P} \cup \mathcal{R} \text{ is top- or roof-raise-bounded, or} \\ & \text{non-duplicating and match-raise-bounded} \\ & \text{for } \mathcal{T}(\mathcal{F}) \\ \{(\mathcal{P},\mathcal{R},\mathcal{G})\} & \text{otherwise} \end{cases}$$

where $\mathcal{F}$ is the signature of $\mathcal{P} \cup \mathcal{R}$, is sound and complete.

Proof. Analogues to the proof of Theorem 5.2 by using Theorem 4.15 instead of Theorem 4.4. □

Similar as in the case of e-bounds, e-DP-bounds can only be used for DP problems $(\mathcal{P},\mathcal{R},\mathcal{G})$ consisting of left-linear TRSs $\mathcal{P}$ and $\mathcal{R}$. The reason is that without left-linearity, rewrite sequences in $(\mathcal{P},\mathcal{R},\mathcal{G})$ cannot be lifted to sequences in $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$ (compare Lemma 5.7). As described in Section 4.3 one can solve that problem by considering the relation $\xrightarrow{\mathsf{r}}_{e\text{-}\mathsf{DP}(\mathcal{P},s\to t,\mathcal{R})}$ which uses raise-rules to deal with non-left-linear rewrite rules. Here the relation $\xrightarrow{\mathsf{r}}_{e\text{-}\mathsf{DP}(\mathcal{P},s\to t,\mathcal{R})}$ is obtained from $\xrightarrow{\mathsf{r}}_{e(\mathcal{R})}$ by replacing the TRS $e(\mathcal{R})$ with $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$ in Definition 4.10.

Definition 5.15. Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a DP problem and $s \to t \in \mathcal{P}$ a rewrite rule. We call $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ *e-raise-DP-bounded* for $s \to t$ and a language L if there exists a number $c \in \mathbb{N}$ such that the height of function symbols occurring in terms in $\xrightarrow{\mathsf{r}}^*_{e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})}(\mathsf{lift}_0(L))$ is at most c.

Note that for left-linear DP problems, e-raise-DP-boundedness coincides with e-DP-boundedness. An immediate consequence of the next lemma is that every derivation according to the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ can be simulated using the rewrite relation $\xrightarrow{\mathsf{r}}_{e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})}$.

Lemma 5.16. *Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a DP problem and $s \to t \in \mathcal{P}$. If $u \to_{s \to t} v$ or $u \to_{\mathcal{S}_{s \to t}} v$ then for all terms u' with $\mathsf{base}(u') = u$ there exists a term v' such that $\mathsf{base}(v') = v$ and $u' \xrightarrow{\mathsf{r}}_{e(s \to t)} v'$ or $u' \xrightarrow{\mathsf{r}}_{e\text{-}\mathsf{DP}(\mathcal{S}_{s \to t})} v'$.*

Proof. Straightforward. ☐

The following two results correspond to Lemmata 5.8 and 5.10.

Lemma 5.17. *Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a DP problem, $s \to t \in \mathcal{P}$, and $c \geqslant 0$. The TRS $\mathsf{top}\text{-}\mathsf{DP}_c(\mathcal{P}, s \to t, \mathcal{R})$ does not admit $\xrightarrow{\mathsf{r}}$ rewrite sequences with infinitely many root $\xrightarrow{\mathsf{r}}_{\mathsf{top}(s \to t)}$-steps.*

Proof. Similar to the proof of Lemma 5.8, using the relation $\xrightarrow{\mathsf{r}}_{\mathsf{top}\text{-}\mathsf{DP}(\mathcal{S}_{s \to t})}$ instead of $\to_{\mathsf{top}\text{-}\mathsf{DP}(\mathcal{S}_{s \to t})}$, $\xrightarrow{\mathsf{r}}_{\mathsf{top}(s \to t)}$ instead of $\xrightarrow{\epsilon}_{\mathsf{top}(s \to t)}$, and Lemma 5.16 instead of Lemma 5.7. ☐

Lemma 5.18. *Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a DP problem, $s \to t \in \mathcal{P}$, and $c \geqslant 0$. If $\mathcal{P} \cup \mathcal{R}$ is non-duplicating then the TRS $\mathsf{match}\text{-}\mathsf{DP}_c(\mathcal{P}, s \to t, \mathcal{R})$ does not admit minimal $\xrightarrow{\mathsf{r}}$ rewrite sequences with infinitely many root $\xrightarrow{\mathsf{r}}_{\mathsf{match}(s \to t)}$-steps.*

Proof. Straightforward adaption of the proof of Lemma 5.10 given in the Appendix B.2. ☐

Since we can simulate every minimal rewrite sequence by a height increasing infinite $\xrightarrow{\mathsf{r}}_{\mathsf{top}\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})}$ and $\xrightarrow{\mathsf{r}}_{\mathsf{match}\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})}$ rewrite sequence we can remove rewrite rules from the TRS $\mathcal{P}$ for which $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is top-raise-DP-bounded or match-raise-DP-bounded.

Theorem 5.19. *Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a DP problem, $s \to t \in \mathcal{P}$, and L a language. If $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is top-raise-DP-bounded for $s \to t$ and L then $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite on L if and only if $(\mathcal{P}, \mathcal{R}, \mathcal{G}) \setminus \{s \to t\}$ is finite on L. If $\mathcal{P}$ and $\mathcal{R}$ are non-duplicating and $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is match-raise-DP-bounded for $s \to t$ and L then $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite on L if and only if $(\mathcal{P}, \mathcal{R}, \mathcal{G}) \setminus \{s \to t\}$ is finite on L.*

Proof. The only-if direction is trivial. For the if direction, suppose that the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G}) \setminus \{s \to t\}$ is finite on L. If $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is not finite on L then there exists a minimal rewrite sequence

$$s_1 \xrightarrow{\epsilon}_{s \to t} t_1 \to^*_{\mathcal{S}_{s \to t}} s_2 \xrightarrow{\epsilon}_{s \to t} t_2 \to^*_{\mathcal{S}_{s \to t}} s_3 \xrightarrow{\epsilon}_{s \to t} \cdots$$

with $s_1 \in L$. Here $e \in \{\mathsf{top}, \mathsf{match}\}$. By Lemma 5.16, this rewrite sequence can be lifted to an infinite $\xrightarrow{\mathsf{r}}_{e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})}$ rewrite sequence starting from $\mathsf{lift}_0(s_1)$.

Since the original sequence contains infinitely many $\xrightarrow{\epsilon}_{s\to t}$-steps the lifted sequence contains infinitely many root $\xrightarrow{\mathsf{r}}_{e(s\to t)}$-steps. Moreover, because $(\mathcal{P},\mathcal{R},\mathcal{G})$ is e-raise-DP-bounded for L, there is a $c \geqslant 0$ such that the height of every function symbol occurring in a term in the lifted sequence is at most c. Hence the employed rules must come from $e\text{-}\mathsf{DP}_c(\mathcal{P}, s \to t, \mathcal{R})$ and therefore $e\text{-}\mathsf{DP}_c(\mathcal{P}, s \to t, \mathcal{R})$ contains a minimal $\xrightarrow{\mathsf{r}}$ rewrite sequence consisting of infinitely many root $\xrightarrow{\mathsf{r}}_{e(s\to t)}$-steps. This however is excluded by Lemmata 5.17 and 5.18. □

Theorem 5.20. *The DP processor*

$$(\mathcal{P},\mathcal{R},\mathcal{G}) \mapsto \begin{cases} \{(\mathcal{P},\mathcal{R},\mathcal{G}) \setminus \{s \to t\}\} & \textit{if } (\mathcal{P},\mathcal{R},\mathcal{G}) \textit{ is top-raise-DP-bounded} \\ & \textit{or non-duplicating and match-raise-} \\ & \textit{DP-bounded for } s \to t \textit{ and } \mathcal{T}(\mathcal{F}) \\ \{(\mathcal{P},\mathcal{R},\mathcal{G})\} & \textit{otherwise} \end{cases}$$

where $\mathcal{F}$ is the signature of $\mathcal{P} \cup \mathcal{R}$, is sound and complete.

Proof. Immediate consequence of Theorem 5.19. □

5.2.3 Automation

To prove automatically that a DP problem is e(-raise)-DP-bounded for some language L we use (quasi-deterministic, raise-consistent, and) compatible tree automata as defined in Chapter 3 and 4.

Lemma 5.21. *Let $(\mathcal{P},\mathcal{R},\mathcal{G})$ be a left-linear DP problem, $s \to t \in \mathcal{P}$, and L a language. Let $\mathcal{A}$ be a tree automaton. If $\mathcal{A}$ is compatible with $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$ and $\mathsf{lift}_0(L)$ then $(\mathcal{P},\mathcal{R},\mathcal{G})$ is e-DP-bounded for $s \to t$ and L.*

Proof. Easy consequence of Theorem 3.3. □

In case of non-left-linear TRSs we obtain the following result.

Lemma 5.22. *Let $(\mathcal{P},\mathcal{R},\mathcal{G})$ be a DP problem, $s \to t \in \mathcal{P}$, and L a language. Let $\mathcal{A}$ be a quasi-deterministic and raise-consistent tree automaton. If $\mathcal{A}$ is compatible with $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$ and $\mathsf{lift}_0(L)$ then $(\mathcal{P},\mathcal{R},\mathcal{G})$ is e-raise-DP-bounded for $s \to t$ and L.*

Proof. Similar as the proof of Theorem 4.24 if we take $\mathcal{F}$ to be the signature of $\mathcal{P} \cup \mathcal{R}$ and replace $e(\mathcal{R})$ by $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$. □

The general procedure for constructing a (quasi-deterministic and raise-consistent) tree automaton that is compatible with $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$ and some language $\mathsf{lift}_0(L)$ is identical to the procedure described in Section 4.4. That means, we solve violations of the compatibility requirement until a (quasi-deterministic, raise-consistent, and) compatible tree automaton is obtained. Thereby, missing paths are established in the same way as if we would check for e(-raise)-boundedness. To ensure that the constructed tree automaton is raise-consistent and hence closed under the implicit raise-steps caused by the rewrite relation $\xrightarrow{\mathsf{r}}$ if $\mathcal{P} \cup \mathcal{R}$ is non-left-linear, one of the approaches described in Subsection 4.4.2 can be applied.

Example 5.23. We show that the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ of Example 5.4 is match-DP-bounded for $s \to t$ by constructing a compatible tree automaton. Here $s \to t$ corresponds to the rewrite rule $\mathsf{F}(\mathsf{g}(x), y) \to \mathsf{H}(x, y)$. As starting point we consider the initial tree automaton consisting of the final state 2 and the transitions

$$\begin{array}{ccc} \mathsf{a}_0 \to 1 & \mathsf{f}_0(1,1) \to 1 & \mathsf{g}_0(1) \to 1 \\ \mathsf{h}_0(1,1) \to 1 & \mathsf{F}_0(1,1) \to 2 & \mathsf{H}_0(1,1) \to 2 \end{array}$$

which accepts the set of all ground terms that have F_0 or H_0 as root symbol and a_0, f_0, g_0, and h_0 below the root. Since $\mathsf{F}_0(\mathsf{g}_0(x), y) \to_{\mathsf{match}(s \to t)} \mathsf{H}_1(x, y)$ and $\mathsf{F}_0(\mathsf{g}_0(1), 1) \to^* 2$, we add the transition $\mathsf{H}_1(1,1) \to 2$. Next we consider $\mathsf{H}_1(x, y) \to_{\mathsf{match\text{-}DP}(\mathcal{P} \setminus \{s \to t\})} \mathsf{F}_1(x, \mathsf{g}_1(y))$ with $\mathsf{H}_1(1,1) \to 2$. By adding the transitions $\mathsf{F}_1(1,3) \to 2$ and $\mathsf{g}_1(1) \to 3$ this compatibility violation is solved. After that the rewrite rule $\mathsf{F}_1(\mathsf{g}_0(x), y) \to_{\mathsf{match}(s \to t)} \mathsf{H}_1(x, y)$ and the derivation $\mathsf{F}_1(\mathsf{g}_0(1), 3) \to^* 2$ give rise to the transition $\mathsf{H}_1(1,3) \to 2$. Finally we have $\mathsf{H}_1(x, y) \to_{\mathsf{match\text{-}DP}(\mathcal{P} \setminus \{s \to t\})} \mathsf{F}_1(x, \mathsf{g}_1(y))$ and $\mathsf{H}_1(1,3) \to 2$. In order to ensure $\mathsf{F}_1(1, \mathsf{g}_1(3)) \to^* 2$ we reuse the transition $\mathsf{F}_1(1,3) \to 2$ and add the new transition $\mathsf{g}_1(3) \to 3$. After that step, the obtained tree automaton is compatible with $\mathsf{match\text{-}DP}(\mathcal{P}, s \to t, \mathcal{R})$. Hence the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is match-DP-bounded for $\mathsf{F}(\mathsf{g}(x), y) \to \mathsf{H}(x, y)$ by 1. Applying the DP processor of Theorem 5.12 yields the new DP problem $(\mathcal{P}', \mathcal{R}, \mathcal{G}')$ with $\mathcal{P}' = \{\mathsf{H}(x, y) \to \mathsf{F}(x, \mathsf{g}(y))\}$ and $\mathcal{G}' = \mathcal{G} \setminus (\mathcal{P} \setminus \mathcal{P}')$, which is easily (and automatically by numerous DP processors) shown to be finite. We note that the DP processor of Theorem 5.2 fails on $(\mathcal{P}, \mathcal{R}, \mathcal{G})$.

Similar as for e(-raise)-bounds we can optimize the completion procedure by constructing a (quasi-deterministic and raise-consistent) tree-automaton that is *quasi-compatible* with $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$ and $\mathsf{lift}_0(L)$.

Lemma 5.24. *Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a DP problem, $s \to t \in \mathcal{P}$, and L a language. Let $\mathcal{A}$ be a tree automaton. If $\mathcal{P}$ and $\mathcal{R}$ are left-linear and $\mathcal{A}$ is quasi-compatible with $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$ and $\mathsf{lift}_0(L)$ then $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is e-DP-bounded for $s \to t$ and L. If $\mathcal{A}$ is quasi-deterministic, raise-consistent, and quasi-compatible with $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$ and $\mathsf{lift}_0(L)$ then $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is e-raise-DP-bounded for $s \to t$ and L.*

Proof. Similar as the proofs of Theorems 4.29 and 4.31; just replace the TRS $e(\mathcal{R})$ by $e\text{-}\mathsf{DP}(\mathcal{P}, s \to t, \mathcal{R})$. ☐

5.2.4 Forward Closures

As mention in Section 4.5, when proving the termination of a TRS $\mathcal{R}$ that is non-overlapping or right-linear it is sufficient to restrict our attention to the set $\mathsf{RFC}_{\mathsf{rhs}(\mathcal{R})}(\mathcal{R})$ of right-hand sides of forward closures. If we want to prove the termination of $\mathcal{R}$ using dependency pairs, we can benefit from the properties of DP problems.

Lemma 5.25. *Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a DP problem. If $\mathcal{P}$ and $\mathcal{R}$ are right-linear then $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite if and only if it is finite on $\mathsf{RFC}_{\mathsf{rhs}(\mathcal{P})}(\mathcal{P} \cup \mathcal{R})$.*

Proof. Easy consequence of Theorem 4.35 and the definition of finiteness. □

Lemma 5.25 can be used in connection with the DP processors of Theorems 5.2 and 5.14. For the DP processors of Theorems 5.12 and 5.20 we can do better. Since the proof is considerably more complicated than the previous one it is deferred to Appendix B.3.

Lemma 5.26. *Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a DP problem and $s \to t \in \mathcal{P}$. If $\mathcal{P}$ and $\mathcal{R}$ are right-linear then $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ admits a minimal rewrite sequence with infinitely many $\xrightarrow{\epsilon}_{s \to t}$-steps if and only if it admits such a sequence starting from a term in $\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R})$.*

In order to make use of the above lemma, we have to construct the set $\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R})$. As explained in Section 4.5 we want to do that by using tree automata completion. To construct a suitable tree automaton we use the TRS $(\mathcal{P} \cup \mathcal{R})_\#$ if $\mathcal{P} \cup \mathcal{R}$ is linear and the TRS $(\mathcal{P} \cup \mathcal{R})'_\#$ if $\mathcal{P} \cup \mathcal{R}$ is right-linear but not left-linear. In the case that $\mathcal{P} \cup \mathcal{R}$ is linear the following results can be derived using Lemma 4.37.

Theorem 5.27. *Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a linear DP problem. If $\mathcal{P} \cup \mathcal{R}$ is match-bounded for $\to^*_{(\mathcal{P} \cup \mathcal{R})_\#}(\mathsf{rhs}(\mathcal{P})\sigma_\#)$ then $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite.*

Proof. Since $\to^*_{(\mathcal{P} \cup \mathcal{R})_\#}(\mathsf{rhs}(\mathcal{P})\sigma_\#)$ is a superset of $\mathsf{RFC}_{\mathsf{rhs}(\mathcal{P})}(\mathcal{P} \cup \mathcal{R})\sigma_\#$, $\mathcal{P} \cup \mathcal{R}$ is also match-raise-bounded for $\mathsf{RFC}_{\mathsf{rhs}(\mathcal{P})}(\mathcal{P} \cup \mathcal{R})\sigma_\#$. (Recall that for linear $\mathcal{P}$ and $\mathcal{R}$, match-boundedness coincides with match-raise-boundedness.) Theorem 4.15 yields the termination of $\mathcal{P} \cup \mathcal{R}$ on $\mathsf{RFC}_{\mathsf{rhs}(\mathcal{P})}(\mathcal{P} \cup \mathcal{R})\sigma_\#$. Since rewriting is closed under substitutions, $\mathcal{P} \cup \mathcal{R}$ is terminating on $\mathsf{RFC}_{\mathsf{rhs}(\mathcal{P})}(\mathcal{P} \cup \mathcal{R})$ and hence the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite on $\mathsf{RFC}_{\mathsf{rhs}(\mathcal{P})}(\mathcal{P} \cup \mathcal{R})$. Applying Lemma 5.25 yields the finiteness of $(\mathcal{P}, \mathcal{R}, \mathcal{G})$. □

Theorem 5.28. *Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a linear DP problem and $s \to t \in \mathcal{P}$ a rewrite rule. If $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is match-DP-bounded for $s \to t$ and $\to^*_{(\mathcal{P} \cup \mathcal{R})_\#}(\{t\}\sigma_\#)$ then the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite if and only if $(\mathcal{P}, \mathcal{R}, \mathcal{G}) \setminus \{s \to t\}$ is finite.*

Proof. Since $\to^*_{(\mathcal{P} \cup \mathcal{R})_\#}(t\sigma_\#)$ is a superset of $\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R})\sigma_\#$, $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is also match-raise-DP-bounded for $s \to t$ and $\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R})\sigma_\#$. (Recall that for linear $\mathcal{P}$ and $\mathcal{R}$, match-DP-boundedness coincides with match-raise-DP-boundedness.) Theorem 5.19 yields that $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite on $\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R})\sigma_\#$ if and only if $(\mathcal{P}, \mathcal{R}, \mathcal{G}) \setminus \{s \to t\}$ is finite on $\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R})\sigma_\#$. Since rewriting is closed under substitutions, $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite on $\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R})$ if and only if $(\mathcal{P}, \mathcal{R}, \mathcal{G}) \setminus \{s \to t\}$ is finite on $\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R})$. From Lemma 5.26 we conclude that $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite if and only if $(\mathcal{P}, \mathcal{R}, \mathcal{G}) \setminus \{s \to t\}$ is finite. □

The above results extend to non-left-linear and right-linear DP problems without any problems. To conclude that $\to^*_{(\mathcal{P} \cup \mathcal{R})'_\#}(L\sigma_\#) \supseteq \mathsf{RFC}_L(\mathcal{P} \cup \mathcal{R})\sigma_\#$ we use Lemma 4.42 instead of Lemma 4.37.

Theorem 5.29. *Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a right-linear DP problem. If $\mathcal{P} \cup \mathcal{R}$ is match-raise-bounded for $\to^*_{(\mathcal{P} \cup \mathcal{R})'_\#}(\mathsf{rhs}(\mathcal{P})\sigma_\#)$ then $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite.*

Proof. Identical to the proof of Theorem 5.27 after replacing $(\mathcal{P} \cup \mathcal{R})_{\#}$ by $(\mathcal{P} \cup \mathcal{R})'_{\#}$. □

Theorem 5.30. *Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a right-linear DP problem and $s \to t \in \mathcal{P}$. If $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is match-raise-DP-bounded for $s \to t$ and $\to^*_{(\mathcal{P} \cup \mathcal{R})'_{\#}}(\{t\}\sigma_{\#})$ then $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite if and only if $(\mathcal{P}, \mathcal{R}, \mathcal{G}) \setminus \{s \to t\}$ is finite.*

Proof. Similar as the proof of Theorem 5.28; just replace $(\mathcal{P} \cup \mathcal{R})_{\#}$ by the TRS $(\mathcal{P} \cup \mathcal{R})'_{\#}$. □

5.3 Beyond Dependency Graphs

One of the most important and frequently used DP processors is the so called *dependency graph processor*. It enables the decomposition of DP problems into smaller subproblems by determining which dependency pairs can follow each other in infinite rewrite sequences. Since the real dependency pair graph cannot be computed, the dependency graph processor requires the computation of an over-approximation of the real dependency graph. In the literature several such approximations are proposed. Arts and Giesl [1] gave an effective algorithm based on abstraction and unification. Kusakari and Toyama [45, 46] employed Huet and Lévy's notion of ω-reduction to approximate dependency graphs for AC-termination. Middeldorp [48] advocated the use of tree automata techniques and in [49] improved the approximation of [1] by taking symmetry into account. Giesl, Thiemann, and Schneider-Kamp [28] tightly coupled abstraction and unification, resulting in an improvement of [1] which is especially suited for applicative systems.

In the following we show that tree automata completion is much more effective for approximating dependency graphs than the method proposed in [48] which approximates the underlying rewrite system to ensure regularity preservation. We further show that by incorporating right-hand sides of forward closures, we can eliminate arcs from the real dependency graph.

5.3.1 Using Dependency Graphs

As mentioned before, the dependency graph processor enables to decompose a DP problem into smaller subproblems. Following [55], we find it convenient to split the processor into one which computes the dependency graph and one which performs the decomposition into smaller subproblems by computing the strongly connected components of the underlying graph. This separates the part that needs to be approximated from the computable part and is important to properly describe the experiments in Section 7.2 where we combine several graph approximations before computing strongly connected components.

Definition 5.31. Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a DP problem. The *dependency graph processor* is defined as

$$(\mathcal{P}, \mathcal{R}, \mathcal{G}) \mapsto \{(\mathcal{P}, \mathcal{R}, \mathcal{G} \cap \mathsf{DG}(\mathcal{P}, \mathcal{R}))\}$$

where $\mathsf{DG}(\mathcal{P}, \mathcal{R})$ is the *dependency graph* of $\mathcal{P}$ and $\mathcal{R}$, which has the rules in $\mathcal{P}$ as nodes and there is an arc from $s \to t$ to $u \to v$ if and only if there exist substitutions σ and τ such that $t\sigma \to_{\mathcal{R}}^{*} u\tau$.

To split a DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ into a set of subproblems, we compute the strongly connected components of the graph $\mathcal{G}$. Thereby each strongly connected component gives rise to one subproblem. Let $\mathcal{G} = (N, E)$ be a graph. A sequence $\alpha_1, \ldots, \alpha_n$ of nodes is called a *path* from α_1 to α_n if $(\alpha_i, \alpha_{i+1}) \in E$ for all $i \in \{1, \ldots, n-1\}$. Let α and β be two nodes in N. We say that α and β are *connected* if there are non-empty paths from α to β and from β to α. A set $M \subseteq N$ of nodes is called a *strongly connected component* (SCC for short) if for all nodes $\alpha, \beta \in M$, α and β are connected and for all nodes $\alpha' \in N \setminus M$ there is a node $\beta' \in M$ such that α' and β' are not connected.

Definition 5.32. For a DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$, the *SCC processor* accomplish the transformation

$$(\mathcal{P}, \mathcal{R}, \mathcal{G}) \mapsto \{(\mathcal{P}_1, \mathcal{R}, \mathcal{G}_1), \ldots, (\mathcal{P}_n, \mathcal{R}, \mathcal{G}_n)\}$$

where $\mathcal{P}_1, \ldots, \mathcal{P}_n$ are the SCCs of the graph $\mathcal{G}$ and $\mathcal{G}_i$ with $i \in \{1, \ldots, n\}$ denotes the subgraph $\mathcal{G} \setminus (\mathcal{P} \setminus \mathcal{P}_i)$ of $\mathcal{G}$.

The following results are well-known [1, 27, 32, 55].

Theorem 5.33. *The dependency graph and SCC processors are sound and complete.* □

Let us illustrate the above definitions and results on an example.

Example 5.34. Consider the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ with $\mathcal{R}$ consisting of the rewrite rules $\mathsf{f}(\mathsf{g}(x), y) \to \mathsf{g}(\mathsf{h}(x, y))$ and $\mathsf{h}(\mathsf{g}(x), y) \to \mathsf{f}(\mathsf{g}(\mathsf{a}), \mathsf{h}(x, y))$, $\mathcal{P}$ consisting of the dependency pairs

$$\begin{array}{ll} 1\colon \mathsf{F}(\mathsf{g}(x), y) \to \mathsf{H}(x, y) & 2\colon \mathsf{H}(\mathsf{g}(x), y) \to \mathsf{F}(\mathsf{g}(\mathsf{a}), \mathsf{h}(x, y)) \\ & 3\colon \mathsf{H}(\mathsf{g}(x), y) \to \mathsf{H}(x, y) \end{array}$$

of $\mathcal{R}$, and $\mathcal{G} = \mathcal{P} \times \mathcal{P}$. Because the left-hand side $\mathsf{H}(\mathsf{g}(x), y)$ is an instance of the right-hand side $\mathsf{H}(x, y)$ and $\mathsf{F}(\mathsf{g}(\mathsf{a}), \mathsf{h}(x, y))$ is an instance of $\mathsf{F}(\mathsf{g}(x), y)$, the graph $\mathsf{DG}(\mathcal{P}, \mathcal{R})$ has five arcs:

$$\begin{tikzcd}
1 \arrow[r, bend left] \arrow[rr, bend left=40] & 2 \arrow[l, bend left] & 3 \arrow[l] \arrow[loop above]
\end{tikzcd}$$

The dependency graph processor of Definition 5.31 returns the new DP problem $(\mathcal{P}, \mathcal{R}, \mathsf{DG}(\mathcal{P}, \mathcal{R}))$. Because the given dependency pairs form a single SCC in the graph $\mathsf{DG}(\mathcal{P}, \mathcal{R})$, the SCC processor of Definition 5.32 does not make progress. Hence we end up with the DP problem $(\mathcal{P}, \mathcal{R}, \mathsf{DG}(\mathcal{P}, \mathcal{R}))$.

5.3.2 Estimating Dependency Graphs

For two TRSs $\mathcal{P}$ and $\mathcal{R}$ the dependency graph $\mathsf{DG}(\mathcal{P},\mathcal{R})$ contains an arc from a dependency pair α to a dependency pair β if and only if there exist substitutions σ and τ such that $\mathsf{rhs}(\alpha)\sigma \to_{\mathcal{R}}^{*} \mathsf{lhs}(\beta)\tau$. Without loss of generality we may assume that $\mathsf{rhs}(\alpha)\sigma$ and $\mathsf{lhs}(\beta)\tau$ are ground terms. Hence there is no arc from α to β if and only if $\Sigma(\mathsf{lhs}(\beta)) \cap \to_{\mathcal{R}}^{*}(\Sigma(\mathsf{rhs}(\alpha))) = \varnothing$. Here $\Sigma(t)$ denotes the set of ground instances of the term t with respect to the signature $\mathcal{G}$ consisting of all function symbols that appear in $\mathcal{P} \cup \mathcal{R}$ minus the root symbols of the left- and right-hand sides of $\mathcal{P}$ that do neither occur on positions below the root in $\mathcal{P}$ nor in $\mathcal{R}$.[2] For an arbitrary term t and regular language L it is decidable whether $\Sigma(t) \cap L = \varnothing$—a result of Tison (see [48])—and hence we can check the above condition by constructing a tree automaton that accepts $\to_{\mathcal{R}}^{*}(\Sigma(\mathsf{rhs}(\alpha)))$. Since this set is in general not regular, we compute an over-approximation with the help of tree automata completion starting from an automaton that accepts $\Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$. Here ren is the function that linearizes its argument by replacing all occurrences of variables with fresh variables. This step is necessary to ensure the regularity of $\Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$.

Definition 5.35. Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs, $\alpha, \beta \in \mathcal{P}$, and L a language. We say that β is *unreachable* from α with respect to L if there is a tree automaton $\mathcal{A}$ compatible with $\mathcal{R}$ and $L \cap \Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$ such that $\mathcal{A}$ is quasi-deterministic whenever $\mathcal{P}$ or $\mathcal{R}$ is non-left-linear and $\Sigma(\mathsf{lhs}(\beta)) \cap \mathcal{L}(\mathcal{A}) = \varnothing$.

The language L in the above definition allows us to refine the set of starting terms $\Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$ which are considered in the computation of an arc from α to β. In Subsection 5.3.3 we make use of the set L to remove arcs from the (real) dependency graph. In the remainder of this subsection we always have $L = \Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$.

Definition 5.36. The nodes of the *c-dependency graph* $\mathsf{DG_c}(\mathcal{P},\mathcal{R})$ are the rewrite rules of $\mathcal{P}$ and there is no arc from α to β if and only if β is unreachable from α with respect to $\Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$.

The c in the above definition refers to the fact that a $\underline{\text{c}}$ompatible tree automaton is $\underline{\text{c}}$onstructed by tree automata $\underline{\text{c}}$ompletion.

Lemma 5.37. *Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. Then* $\mathsf{DG_c}(\mathcal{P},\mathcal{R}) \supseteq \mathsf{DG}(\mathcal{P},\mathcal{R})$.

Proof. Easy consequence of Theorems 3.3 and 3.16. □

As explained in Chapter 3 we try to construct a (quasi-deterministic) tree automata $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ that is compatible with the TRS $\mathcal{R}$ and the language $\Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$ by resolving violations of the compatibility requirement: If $l\sigma \to_{\Delta}^{*} q$ ($l\sigma \to_{\Delta_\phi}^{*} q$) but not $r\sigma \to_{\Delta}^{*} q$ for some rewrite rule $l \to r \in \mathcal{R}$, state substitution $\sigma\colon \mathcal{V}\mathsf{ar}(l) \to Q$ ($\sigma\colon \mathcal{V}\mathsf{ar}(l) \to Q_\phi$), and state $q \in Q$, then we add new states and transitions to the current tree automaton to ensure

[2]If each function symbol in $\mathcal{P} \cup \mathcal{R}$ is identical to the root symbol of a left- or right-hand side of $\mathcal{P}$, then we require that $\mathcal{G}$ contains a fresh constant $\#$. This constant is added to the signature to ensure that $\Sigma(t)$ cannot be empty.

that $r\sigma \rightarrow^*_\Delta q$. Note that in contrast to the match-bound technique we can always compute a compatible tree automaton if the considered TRS as well as its signature are finite.

Example 5.38. We consider the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ of Example 5.34. The tree automaton $\mathcal{A}$ consisting of the final state 2 and the transitions

$$\mathsf{a} \rightarrow 1 \qquad \mathsf{f}(1,1) \rightarrow 1 \qquad \mathsf{g}(1) \rightarrow 1 \qquad \mathsf{h}(1,1) \rightarrow 1 \qquad \mathsf{H}(1,1) \rightarrow 2$$

accepts the language $\Sigma(\mathsf{H}(x,y))$ and is easily constructed. Because the automaton $\mathcal{A}$ is already compatible with $\mathcal{R}$ and $\Sigma(\mathsf{H}(x,y))$, completion is trivial here. So we have $\mathcal{L}(\mathcal{A}) = \rightarrow^*_\mathcal{R}(\Sigma(\mathsf{H}(x,y))) = \Sigma(\mathsf{H}(x,y))$. As $\mathsf{H}(\mathsf{g}(\mathsf{a}), \mathsf{a})$ is accepted by $\mathcal{A}$, $\mathsf{DG_c}(\mathcal{P}, \mathcal{R})$ contains arcs from 1 and 3 to 2 and 3. Similarly, we can construct a tree automaton $\mathcal{B}$ which is compatible with $\mathcal{R}$ and $\Sigma(\mathsf{F}(\mathsf{g}(\mathsf{a}), \mathsf{h}(x,y)))$:

$$\mathsf{a} \rightarrow 1 \mid 2 \quad \mathsf{f}(1,1) \rightarrow 1 \mid 4 \quad \mathsf{g}(1) \rightarrow 1 \mid 4 \quad \mathsf{h}(1,1) \rightarrow 1 \mid 4 \quad \mathsf{F}(3,4) \rightarrow 2$$
$$\mathsf{g}(2) \rightarrow 3$$

Because $\mathsf{F}(\mathsf{g}(\mathsf{a}), \mathsf{h}(\mathsf{a}, \mathsf{a}))$ is accepted by $\mathcal{B}$, we obtain an arc from 2 to 1. Further arcs do not exist.

It can be argued that the use of tree automata techniques for the DP problem of Example 5.34 is a waste of resources because the dependency graph can also be computed by just taking the root symbols of the dependency pairs into consideration. However, in the next subsection we show that this radically changes when taking right-hand sides of forward closures into account.

5.3.3 Incorporating Forward Closures

Given a DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ and two rewrite rules $\alpha, \beta \in \mathcal{P}$, an arc from α to β in the dependency graph $\mathsf{DG}(\mathcal{P}, \mathcal{R})$ is an indication that β may follow α in an *infinite* sequence in $\mathcal{P} \cup \mathcal{R}$. However it is not a sufficient condition because if the problem is finite there are no infinite sequences whatsoever. What we would like to determine is whether β can follow α infinitely many times in a minimal sequence. To express this property we use the notion defined below.

Definition 5.39. Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a DP problem and $\alpha, \beta \in \mathcal{P}$. We say that *$\beta$ directly follows α* in a minimal sequence

$$s_1 \xrightarrow{\epsilon}_{\alpha_1} t_1 \rightarrow^*_\mathcal{R} s_2 \xrightarrow{\epsilon}_{\alpha_2} t_2 \rightarrow^*_\mathcal{R} \cdots$$

if $\alpha_i = \alpha$ and $\alpha_{i+1} = \beta$ for some $i \geqslant 1$.

With the existing approximations of the dependency graph, only local connections can be tested. In this subsection we show that by using right-hand sides of forward closures, we can sometimes delete arcs from the dependency graph which cannot occur infinitely many times in minimal sequences. The key ingredient to this extension is based on the observation that for each minimal rewrite sequence there is a minimal rewrite sequence starting at a term in $\mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R})$ which exhibits a similar structure as the original sequence.

Lemma 5.40. *Let $\mathcal{P}$ and $\mathcal{R}$ be two right-linear TRSs and $\alpha, \beta \in \mathcal{P}$. The TRS $\mathcal{P} \cup \mathcal{R}$ admits a minimal rewrite sequence in which infinitely many β-steps directly follow α-steps if and only if it admits such a sequence starting from a term in $\mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R})$.*

Since the proof of the above lemma is quite complicated and lengthy it is deferred to Appendix B.3. Based on Lemma 5.40 we are now ready to define a new form of dependency graphs.

Definition 5.41. Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. The *improved dependency graph* of $\mathcal{P}$ and $\mathcal{R}$, denoted by $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$, has the rules in $\mathcal{P}$ as nodes and there is an arc from $s \to t$ to $u \to v$ if and only if there exist substitutions σ and τ such that $t\sigma \to_{\mathcal{R}}^* u\tau$ and $t\sigma \in \Sigma_\#(\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R}))$. Here $\Sigma_\#$ is the function that replaces all variables by the fresh constant $\#$.

Note that the use of $\Sigma_\#$ in the above definition is essential. If we would replace $\Sigma_\#$ by Σ then $\Sigma(\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R})) \supseteq \Sigma(t)$ because $t \in \mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R})$ and hence $\mathsf{IDG}(\mathcal{P}, \mathcal{R}) = \mathsf{DG}(\mathcal{P}, \mathcal{R})$. According to the following lemma the improved dependency graph can be used whenever the participating TRSs are right-linear.

Lemma 5.42. *Let $\mathcal{P}$ and $\mathcal{R}$ be right-linear TRSs and $\alpha, \beta \in \mathcal{P}$. If $\mathcal{P} \cup \mathcal{R}$ admits a minimal rewrite sequence in which infinitely many β-steps directly follow α-steps, then $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ admits an arc from α to β.*

Proof. Assume that there is a minimal rewrite sequence

$$s_1 \xrightarrow{\epsilon}_{\mathcal{P}} t_1 \to_{\mathcal{R}}^* s_2 \xrightarrow{\epsilon}_{\mathcal{P}} t_2 \to_{\mathcal{R}}^* s_3 \xrightarrow{\epsilon}_{\mathcal{P}} \cdots$$

in which infinitely many β-steps directly follow α-steps. Due to Lemma 5.40 we may assume without loss of generality that $s_1 \in \mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R})$. Let $i \geqslant 1$ such that $s_i \xrightarrow{\epsilon}_{\alpha} t_i \to_{\mathcal{R}}^* s_{i+1} \xrightarrow{\epsilon}_{\beta} t_{i+1}$. Because $\mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R})$ is closed under rewriting with respect to $\mathcal{P}$ and $\mathcal{R}$ we know that $t_i \in \mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R})$. We have $t_i = \mathsf{rhs}(\alpha)\sigma$ and $s_{i+1} = \mathsf{lhs}(\beta)\tau$ for some substitutions σ and τ. From $t_i \in \mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R})$ we infer that $t_i\theta \in \Sigma_\#(\mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R}))$ for the substitution θ that replaces every variable by $\#$. Due to the fact that rewriting is closed under substitutions we have $t_i\theta \to_{\mathcal{R}}^* s_{i+1}\theta$. Hence $\mathsf{rhs}(\alpha)\sigma\theta \to_{\mathcal{R}}^* \mathsf{lhs}(\beta)\tau\theta$ and therefore $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ contains an arc from α to β. □

The following example shows that it is essential to include $\mathcal{P}$ in the construction of the set $\Sigma_\#(\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R}))$ in the definition of $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$.

Example 5.43. Consider the TRS $\mathcal{R}$ consisting of the three rules $\mathsf{f}(x) \to \mathsf{g}(x)$, $\mathsf{g}(\mathsf{a}) \to \mathsf{h}(\mathsf{b})$, and $\mathsf{h}(x) \to \mathsf{f}(\mathsf{a})$, the TRS $\mathcal{P} = \mathsf{DP}(\mathcal{R})$ consisting of the rules

$$\mathsf{F}(x) \to \mathsf{G}(x) \qquad \mathsf{G}(\mathsf{a}) \to \mathsf{H}(\mathsf{b}) \qquad \mathsf{H}(x) \to \mathsf{F}(\mathsf{a})$$

and the graph $\mathcal{G} = (\mathcal{P}, \mathcal{P} \times \mathcal{P})$. The DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is not finite because it admits the following cycle:

$$\mathsf{F}(\mathsf{a}) \xrightarrow{\epsilon}_{\mathcal{P}} \mathsf{G}(\mathsf{a}) \xrightarrow{\epsilon}_{\mathcal{P}} \mathsf{H}(\mathsf{b}) \xrightarrow{\epsilon}_{\mathcal{P}} \mathsf{F}(\mathsf{a})$$

Let $t = \mathsf{G}(x)$. We have $\mathsf{RFC}_t(\mathcal{R}) = \{t\}$ and hence $\Sigma_\#(\mathsf{RFC}_t(\mathcal{R})) = \{\mathsf{G}(\#)\}$. If we now replace $\Sigma_\#(\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R}))$ by $\Sigma_\#(\mathsf{RFC}_t(\mathcal{R}))$ in Definition 5.41, we would conclude that $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ does not contain an arc from $\mathsf{F}(x) \to \mathsf{G}(x)$ to $\mathsf{G}(\mathsf{a}) \to \mathsf{H}(\mathsf{b})$ because $\mathsf{G}(\#)$ is a normal form which is different from $\mathsf{G}(\mathsf{a})$. But this makes the resulting DP problem $(\mathcal{P}, \mathcal{R}, \mathsf{IDG}(\mathcal{P}, \mathcal{R}))$ finite.

Theorem 5.44. *The* improved dependency graph processor

$$(\mathcal{P}, \mathcal{R}, \mathcal{G}) \mapsto \begin{cases} \{(\mathcal{P}, \mathcal{R}, \mathcal{G} \cap \mathsf{IDG}(\mathcal{P}, \mathcal{R}))\} & \textit{if } \mathcal{P} \cup \mathcal{R} \textit{ is right-linear} \\ \{(\mathcal{P}, \mathcal{R}, \mathcal{G} \cap \mathsf{DG}(\mathcal{P}, \mathcal{R}))\} & \textit{otherwise} \end{cases}$$

is sound and complete.

Proof. Soundness is an easy consequence of Lemma 5.42 and Theorem 5.33. Completeness is guaranteed by the two inclusions $\mathcal{G} \cap \mathsf{DG}(\mathcal{P}, \mathcal{R}) \subseteq \mathcal{G}$ and $\mathcal{G} \cap \mathsf{IDG}(\mathcal{P}, \mathcal{R}) \subseteq \mathcal{G}$. □

Example 5.45. We consider again the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ of Example 5.34. Let $s = \mathsf{H}(x, y)$ and $t = \mathsf{F}(\mathsf{g}(\mathsf{a}), \mathsf{h}(x, y))$. We first compute $\mathsf{RFC}_s(\mathcal{P} \cup \mathcal{R})$ and $\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R})$. The former set consists of $\mathsf{H}(x, y)$ together with all terms in the set $\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R})$. Each term contained in the latter set is an instance of $\mathsf{F}(\mathsf{g}(\mathsf{a}), x)$ or $\mathsf{H}(\mathsf{a}, x)$. It follows that each term in $\Sigma_\#(\mathsf{RFC}_s(\mathcal{P} \cup \mathcal{R}))$ is a ground instance of $\mathsf{F}(\mathsf{g}(\mathsf{a}), x)$ or $\mathsf{H}(\mathsf{a}, x)$ or equal to the term $\mathsf{H}(\#, \#)$. Similarly, each term in $\Sigma_\#(\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R}))$ is a ground instance of $\mathsf{F}(\mathsf{g}(\mathsf{a}), x)$ or $\mathsf{H}(\mathsf{a}, x)$. Hence $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ contains an arc from 2 to 1. Further arcs do not exist because there are no substitution τ and term $u \in \Sigma_\#(\mathsf{RFC}_s(\mathcal{P} \cup \mathcal{R}))$ such that $u \to_{\mathcal{R}}^* \mathsf{H}(\mathsf{g}(x), y)\tau$. So $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ looks as follows:

$$\begin{tikzcd} 1 & 2 \arrow[l] & 3 \end{tikzcd}$$

Hence the improved dependency graph processor produces the new DP problem $(\mathcal{P}, \mathcal{R}, \mathsf{IDG}(\mathcal{P}, \mathcal{R}))$. Since the above graph does not admit any SCCs, the SCC processor returns an empty list of DP problems. Consequently, the TRS $\mathcal{R}$ is terminating.

Similar to $\mathsf{DG}(\mathcal{P}, \mathcal{R})$, $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ is not computable in general. We over-approximate $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ by using tree automata completion as described in Subsection 5.3.2.

Definition 5.46. Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. The nodes of the *c-improved dependency graph* $\mathsf{IDG_c}(\mathcal{P}, \mathcal{R})$ are the rewrite rules of $\mathcal{P}$ and there is no arc from α to β if and only if β is unreachable from α with respect to the language $\Sigma_\#(\mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R}))$.

Lemma 5.47. *Let* $\mathcal{P}$ *and* $\mathcal{R}$ *be two TRSs. Then* $\mathsf{IDG_c}(\mathcal{P}, \mathcal{R}) \supseteq \mathsf{IDG}(\mathcal{P}, \mathcal{R})$.

Proof. Assume to the contrary that the lemma does not hold. Then there are two rewrite rules $s \to t$ and $u \to v$ in $\mathcal{P}$ such that there is an arc from $s \to t$ to $u \to v$ in $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ but not in $\mathsf{IDG_c}(\mathcal{P}, \mathcal{R})$. By Definition 5.41 there are substitutions σ and τ such that $t\sigma \in L$ with $L = \Sigma_\#(\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R}))$ and

$t\sigma \rightarrow_{\mathcal{R}}^{*} u\tau$. Since $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$ does not admit an arc from $s \rightarrow t$ to $u \rightarrow v$, there is a tree automaton $\mathcal{A}$ compatible with $\mathcal{R}$ and $L \cap \Sigma(\mathsf{ren}(t))$ such that $\mathcal{A}$ is quasi-deterministic if $\mathcal{P}$ or $\mathcal{R}$ is non-left-linear and $\Sigma(u) \cap \mathcal{L}(\mathcal{A}) = \varnothing$. Theorems 3.3 and 3.16 yield $\rightarrow_{\mathcal{R}}^{*}(L \cap \Sigma(\mathsf{ren}(t))) \subseteq \mathcal{L}(\mathcal{A})$. From $t\sigma \in L \cap \Sigma(\mathsf{ren}(t))$ and $t\sigma \rightarrow_{\mathcal{R}}^{*} u\tau$ we infer that $u\tau \in \mathcal{L}(\mathcal{A})$, contradicting $\Sigma(u) \cap \mathcal{L}(\mathcal{A}) = \varnothing$. □

To compute $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$ we have to construct an intermediate tree automaton that accepts $\mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R})$. This can be done by using tree automata completion as described in Subsection 5.2.4. That means, we construct a tree automaton $\mathcal{A}$ that is compatible with the TRS $\mathcal{S}$ and the language $\{\mathsf{rhs}(\alpha)\}\sigma_{\#}$. Here $\mathcal{S} = (\mathcal{P} \cup \mathcal{R})_{\#}$ if $\mathcal{P} \cup \mathcal{R}$ is linear and $\mathcal{S} = (\mathcal{P} \cup \mathcal{R})'_{\#}$ if $\mathcal{P} \cup \mathcal{R}$ is right-linear but not left-linear. Afterwards we convert $\mathcal{A}$ into a tree automaton $\mathcal{A}'$ which accepts only those terms in $\mathcal{L}(\mathcal{A})$ that are also contained in $\Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$. Starting from this automaton we can then check whether $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$ admits an arc from α to β by transforming $\mathcal{A}'$ into a tree automaton that is compatible with $\mathcal{R}$, as discussed in Subsection 5.3.2. We continue our leading example.

Example 5.48. We construct $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$ for the DP problem $(\mathcal{P},\mathcal{R},\mathcal{G})$ of Example 5.34. Let $s = \mathsf{H}(x,y)$. The tree automaton $\mathcal{A}$ consisting of the final state 2 and the transitions

$$\begin{array}{lllll}
\# \rightarrow 1 & \mathsf{g}(3) \rightarrow 4 & \mathsf{f}(4,5) \rightarrow 5 & \mathsf{h}(1,1) \rightarrow 5 & \mathsf{H}(1,1) \rightarrow 2 \\
\mathsf{a} \rightarrow 3 & \mathsf{g}(6) \rightarrow 5 & & \mathsf{h}(3,5) \rightarrow 6 & \mathsf{H}(3,5) \rightarrow 2
\end{array}$$

is compatible with $\mathcal{R}$ and $\Sigma_{\#}(\mathsf{RFC}_s(\mathcal{P} \cup \mathcal{R})) \cap \Sigma(s)$. Since $\mathcal{A}$ does not accept any ground instance of the term $\mathsf{H}(\mathsf{g}(x),y)$ we conclude that the rules 2 and 3 are unreachable from 1 and 3. It remains to check whether there is any outgoing arc from rule 2. Let $t = \mathsf{F}(\mathsf{g}(\mathsf{a}),\mathsf{h}(x,y))$ be the right-hand side of 2. Similar as before we can construct a tree automaton $\mathcal{B}$ consisting of the final state 5 and the transitions

$$\begin{array}{lllll}
\# \rightarrow 1 & \mathsf{g}(2) \rightarrow 3 & \mathsf{f}(3,4) \rightarrow 4 & \mathsf{h}(1,1) \rightarrow 4 & \mathsf{F}(3,4) \rightarrow 5 \\
\mathsf{a} \rightarrow 2 & \mathsf{g}(6) \rightarrow 4 & & \mathsf{h}(2,4) \rightarrow 6 &
\end{array}$$

which is compatible with $\mathcal{R}$ and $\Sigma_{\#}(\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R})) \cap \Sigma(t)$. Since the instance $\mathsf{F}(\mathsf{g}(\mathsf{a}),\mathsf{h}(\#,\#))$ of $\mathsf{F}(\mathsf{g}(x),y)$ is accepted by $\mathcal{B}$, $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$ contains an arc from 2 to 1. Further arcs do not exist. Hence $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$ coincides with $\mathsf{IDG}(\mathcal{P},\mathcal{R})$.

5.3.4 Comparison

In the literature several over-approximations of the dependency graph are described [1, 28, 32, 46, 48, 49]. In this subsection we compare the tree automata approach to approximate the processors of Definition 5.31 and Theorem 5.44, developed in the preceding subsections, with the earlier tree automata approach of [48] as well as the approximation used in tools like AProVE [26] and T$_\mathsf{T}$T$_2$ [44], which is a combination of ideas of [28, 49]. We start by formally defining the latter approach.

Let $\mathcal{R}$ be a set of rewrite rules and t a term. The function $\mathsf{tcap}(\mathcal{R},t)$ is defined as $\mathsf{tcap}(\mathcal{R},t) = f(\mathsf{tcap}(\mathcal{R},t_1),\ldots,\mathsf{tcap}(\mathcal{R},t_n))$ if $t = f(t_1,\ldots,t_n)$ and

$f(\mathsf{tcap}(\mathcal{R}, t_1), \ldots, \mathsf{tcap}(\mathcal{R}, t_n))$ does not unify with any $l \in \mathsf{lhs}(\mathcal{R})$. Otherwise $\mathsf{tcap}(\mathcal{R}, t) = x$ for some fresh variable x.

Definition 5.49. Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. The nodes of the *estimated dependency graph* $\mathsf{DG_e}(\mathcal{P}, \mathcal{R})$ are the rewrite rules of $\mathcal{P}$ and there is an arc from $s \to t$ to $u \to v$ if and only if $\mathsf{tcap}(\mathcal{R}, t)$ and u as well as t and $\mathsf{tcap}(\mathcal{R}^{-1}, u)$ are unifiable.

The approach described in [48] to approximate dependency graphs based on tree automata techniques relies on regularity preservation rather than completion. Below we recall the relevant definitions. Let $\mathcal{F}$ be some signature and $\mathcal{R}$ a set of rewrite rules over the signature $\mathcal{F}$.[3] An *approximation mapping* is a mapping ϕ from sets of rewrite rules to sets of rewrite rules such that $\to_{\mathcal{R}} \subseteq \to^*_{\phi(\mathcal{R})}$. We say that ϕ is *regularity preserving* if the set $\leftarrow^*_{\phi(\mathcal{R})}(L) = \{s \in \mathcal{T}(\mathcal{F}) \mid s \to^*_{\phi(\mathcal{R})} t \text{ for some } t \in L\}$ is regular for all sets of rewrite rules $\mathcal{R}$ and regular languages L. The approximation mappings s, nv, and g are defined as follows: $\mathsf{s}(\mathcal{R}) = \{\mathsf{ren}(l) \to x \mid l \to r \in \mathcal{R} \text{ and } x \text{ is a fresh variable}\}$, $\mathsf{nv}(\mathcal{R}) = \{\mathsf{ren}(l) \to \mathsf{ren}(r) \mid l \to r \in \mathcal{R}\}$, and $\mathsf{g}(\mathcal{R})$ is defined as any set of left-linear rewrite rules that is obtained from $\mathcal{R}$ by linearizing the left-hand sides and renaming the variables in the right-hand sides that occur at a depth greater than 1 in the corresponding left-hand sides. These mappings are known to be regularity preserving [10, 51].

Definition 5.50. Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs and let ϕ be an approximation mapping. The nodes of the *ϕ-approximated dependency graph* $\mathsf{DG}_\phi(\mathcal{P}, \mathcal{R})$ are the rewrite rules of $\mathcal{P}$ and there is an arc from $s \to t$ to $u \to v$ if and only if both $\Sigma(t) \cap \leftarrow^*_{\phi(\mathcal{R})}(\Sigma(\mathsf{ren}(u))) \neq \varnothing$ and $\Sigma(u) \cap \leftarrow^*_{\phi(\mathcal{R}^{-1})}(\Sigma(\mathsf{ren}(t))) \neq \varnothing$.

The following results originate from [28, 48].

Lemma 5.51. *For TRSs $\mathcal{P}$ and $\mathcal{R}$, $\mathsf{DG_e}(\mathcal{P}, \mathcal{R}) \supseteq \mathsf{DG}(\mathcal{P}, \mathcal{R})$ and $\mathsf{DG_s}(\mathcal{P}, \mathcal{R}) \supseteq \mathsf{DG_{nv}}(\mathcal{P}, \mathcal{R}) \supseteq \mathsf{DG_g}(\mathcal{P}, \mathcal{R}) \supseteq \mathsf{DG}(\mathcal{P}, \mathcal{R})$.* □

From Examples 5.34 and 5.48 it is obvious that neither $\mathsf{DG_e}(\mathcal{P}, \mathcal{R})$ nor the graph $\mathsf{DG_g}(\mathcal{P}, \mathcal{R})$ subsumes $\mathsf{IDG_c}(\mathcal{P}, \mathcal{R})$. The converse depends very much on the approximation strategy that is used; it can always happen that the completion procedure does not terminate or that the over-approximation is too inexact. Nevertheless, there are problems where $\mathsf{DG_e}(\mathcal{P}, \mathcal{R})$ and $\mathsf{DG_s}(\mathcal{P}, \mathcal{R})$ are properly contained in $\mathsf{IDG_c}(\mathcal{P}, \mathcal{R})$.

Example 5.52. Consider the TRS $\mathcal{R}$ consisting of the rewrite rules $\mathsf{g}(\mathsf{a}, \mathsf{b}) \to \mathsf{b}$, $\mathsf{f}(x, x) \to \mathsf{f}(\mathsf{a}, \mathsf{g}(x, \mathsf{b}))$, and $\mathsf{f}(\mathsf{a}, \mathsf{g}(x, x)) \to \mathsf{f}(\mathsf{a}, \mathsf{a})$, and the TRS $\mathcal{P}$ consisting of the dependency pairs

$$1\colon \mathsf{F}(x, x) \to \mathsf{F}(\mathsf{a}, \mathsf{g}(x, \mathsf{b})) \qquad\qquad 3\colon \mathsf{F}(\mathsf{a}, \mathsf{g}(x, x)) \to \mathsf{F}(\mathsf{a}, \mathsf{a})$$
$$2\colon \mathsf{F}(x, x) \to \mathsf{G}(x, \mathsf{b})$$

[3]To simplify the presentation we assume in the following that whenever we consider two TRSs $\mathcal{P}$ and $\mathcal{R}$ or a DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$, $\mathcal{F}$ represents the signature of $\mathcal{P} \cup \mathcal{R}$, if necessary, extended by the fresh constant $\#$.

of $\mathcal{R}$. First we compute $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$. Because $\mathsf{F}(\mathsf{a},\mathsf{a})$ is an instance of $\mathsf{F}(x,x)$ we obviously have arcs from 3 to 1 and 2. Since $\mathsf{F}(\mathsf{a},\mathsf{g}(\#,\mathsf{b})) \to_{\mathcal{P}'_\#} \mathsf{F}(\mathsf{a},\mathsf{a})$ using the rewrite rule $\mathsf{F}(\mathsf{a},\mathsf{g}(x,y)) \to \mathsf{F}(\mathsf{a},\mathsf{a})$, $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$ contains arcs from 1 to 1 and 2. Moreover, the derivation $\mathsf{F}(\mathsf{a},\mathsf{g}(\#,\mathsf{b})) \to_{\mathcal{R}'_\#} \mathsf{F}(\mathsf{a},\mathsf{b}) \to_{\mathcal{P}'_\#} \mathsf{F}(\mathsf{a},\mathsf{g}(\mathsf{b},\mathsf{b}))$ together with $\mathsf{F}(\mathsf{a},\mathsf{g}(\mathsf{b},\mathsf{b})) \in \Sigma(\mathsf{F}(\mathsf{a},\mathsf{g}(x,x)))$ yields an arc from 1 to 3. Note that in the above rewrite sequence the rewrite rules $\mathsf{g}(\#,\mathsf{b}) \to \mathsf{b}$ and $\mathsf{F}(y,x) \to \mathsf{F}(\mathsf{a},\mathsf{g}(x,\mathsf{b}))$ have been used. Similarly, there is an arc from 3 to 3 because $\mathsf{F}(\mathsf{a},\mathsf{a}) \to_{\mathcal{P}'_\#} \mathsf{F}(\mathsf{a},\mathsf{g}(\mathsf{a},\mathsf{b})) \to_{\mathcal{R}'_\#} \mathsf{F}(\mathsf{a},\mathsf{b}) \to_{\mathcal{P}'_\#} \mathsf{F}(\mathsf{a},\mathsf{g}(\mathsf{b},\mathsf{b}))$ and $\mathsf{F}(\mathsf{a},\mathsf{g}(\mathsf{b},\mathsf{b})) \in \Sigma(\mathsf{F}(\mathsf{a},\mathsf{g}(x,x)))$. Further arcs do not exist. Hence $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$ looks as follows:

$$\begin{tikzcd}
1 \arrow[loop above] \arrow[rr, bend left] \arrow[dr] & & 3 \arrow[loop above] \arrow[ll, bend left] \arrow[dl] \\
& 2 &
\end{tikzcd}$$

In contrast to $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$, the graphs $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$ and $\mathsf{DG_s}(\mathcal{P},\mathcal{R})$ do not admit an arc from 3 to 3. In case of $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$ we have $\mathsf{tcap}(\mathcal{R},\mathsf{F}(\mathsf{a},\mathsf{a})) = \mathsf{F}(\mathsf{a},\mathsf{a})$ and $\mathsf{F}(\mathsf{a},\mathsf{a})$ does not unify with $\mathsf{F}(\mathsf{a},\mathsf{g}(x,x))$. In case of $\mathsf{DG_s}(\mathcal{P},\mathcal{R})$ we have $\mathsf{F}(\mathsf{a},\mathsf{a}) \notin \leftarrow^*_{\mathsf{s}(\mathcal{R})}(\Sigma(\mathsf{ren}(\mathsf{F}(\mathsf{a},\mathsf{g}(x,x)))))$ because $\mathsf{F}(\mathsf{a},\mathsf{a})$ is a normal form with respect to $\mathsf{s}(\mathcal{R})$ and not a ground instance of $\mathsf{ren}(\mathsf{F}(\mathsf{a},\mathsf{g}(x,x)))$.

In case of $\mathsf{DG_c}(\mathcal{P},\mathcal{R})$ we obtain similar results as for $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$. The following example shows that neither $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$ nor $\mathsf{DG_g}(\mathcal{P},\mathcal{R})$ subsumes $\mathsf{DG_c}(\mathcal{P},\mathcal{R})$.

Example 5.53. Consider the TRSs $\mathcal{R}$ and $\mathcal{P}$ with $\mathcal{R}$ consisting of the rewrite rules $\mathsf{p}(\mathsf{p}(\mathsf{p}(x))) \to \mathsf{p}(\mathsf{p}(x))$, $\mathsf{f}(x) \to \mathsf{g}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a}))))$, and $\mathsf{g}(\mathsf{p}(\mathsf{p}(\mathsf{s}(x)))) \to \mathsf{f}(x)$ and $\mathcal{P} = \mathsf{DP}(\mathcal{R})$ consisting of the following rules:

1: $\mathsf{F}(x) \to \mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a}))))$	3: $\mathsf{F}(x) \to \mathsf{P}(\mathsf{p}(\mathsf{a}))$	5: $\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{s}(x)))) \to \mathsf{F}(x)$
2: $\mathsf{F}(x) \to \mathsf{P}(\mathsf{p}(\mathsf{p}(\mathsf{a})))$	4: $\mathsf{F}(x) \to \mathsf{P}(\mathsf{a})$	

First we compute $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$. It is clear that $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$ contains arcs from 5 to 1, 2, 3, and 4. Furthermore, it contains an arc from 1 to 5 because the term $\mathsf{tcap}(\mathcal{R},\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a}))))) = \mathsf{G}(y)$ unifies with $\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{s}(x))))$ and $\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a}))))$ unifies with $\mathsf{tcap}(\mathcal{R}^{-1},\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{s}(x))))) = \mathsf{G}(y)$. Further arcs do not exist and hence $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$ looks as follows:

$$\begin{tikzcd}
1 \arrow[r, bend left] & 5 \arrow[l, bend left] \arrow[r] \arrow[dl] \arrow[dr] & 4 \\
2 & & 3
\end{tikzcd}$$

Next we compute $\mathsf{DG_g}(\mathcal{P},\mathcal{R})$. Similarly as $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$, $\mathsf{DG_g}(\mathcal{P},\mathcal{R})$ has arcs from 5 to 1, 2, 3, and 4. Furthermore $\mathsf{DG_g}(\mathcal{P},\mathcal{R})$ contains an arc from 1 to 5 because $\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a})))) \to_{\mathsf{g}(\mathcal{R})} \mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{s}(\mathsf{a})))) \in \Sigma(\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{s}(x)))))$ by applying the rewrite rule $\mathsf{p}(\mathsf{p}(\mathsf{p}(x))) \to \mathsf{p}(\mathsf{p}(y))$ and $\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{s}(x)))) \to_{\mathsf{g}(\mathcal{R}^{-1})} \mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a}))))$ using the rule $\mathsf{p}(\mathsf{p}(x)) \to \mathsf{p}(\mathsf{p}(\mathsf{p}(y)))$. Hence $\mathsf{DG_g}(\mathcal{P},\mathcal{R})$ coincides with $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$. The graph $\mathsf{DG_c}(\mathcal{P},\mathcal{R})$

$$\begin{tikzcd}
1 & & 5 \arrow[ll] \arrow[rr] \arrow[dl] \arrow[dr] & & 4 \\
& 2 & & 3 &
\end{tikzcd}$$

does not contain an arc from 1 to 5 because 5 is unreachable from 1. This is certified by the tree automaton $\mathcal{A}$ consisting of the final state 5 and the following transitions:

$$\mathsf{a} \to 1 \qquad \mathsf{p}(1) \to 2 \qquad \mathsf{p}(2) \to 3 \mid 4 \qquad \mathsf{p}(3) \to 4 \qquad \mathsf{G}(4) \to 5$$

Note that $\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a})))) \in \mathcal{L}(\mathcal{A})$ and $\Sigma(\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{s}(x))))) \cap \mathcal{L}(\mathcal{A}) = \varnothing$. Furthermore, $\mathcal{L}(\mathcal{A}) = \to_{\mathcal{R}}^{*}(\{\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a}))))\}) = \{\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a})))), \mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{a})))\}$.

Concerning the converse direction, there are TRSs such that $\mathsf{DG}_{\mathsf{e}}(\mathcal{P},\mathcal{R})$ and $\mathsf{DG}_{\mathsf{nv}}(\mathcal{P},\mathcal{R})$ are properly contained in $\mathsf{DG}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$. We assume that this also holds for $\mathsf{DG}_{\mathsf{s}}(\mathcal{P},\mathcal{R})$ although we did not succeed in finding an example.

Example 5.54. Consider the TRS $\mathcal{R}$ consisting of the rule $\mathsf{f}(\mathsf{a},\mathsf{b},x) \to \mathsf{f}(x,x,x)$ and the TRS $\mathcal{P}$ consisting of the rewrite rule $\mathsf{F}(\mathsf{a},\mathsf{b},x) \to \mathsf{F}(x,x,x)$. Obviously, $\mathsf{DG}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$ admits an arc from $\mathsf{F}(\mathsf{a},\mathsf{b},x) \to \mathsf{F}(x,x,x)$ to itself because the ground instance $\mathsf{F}(\mathsf{a},\mathsf{b},\mathsf{a})$ of $\mathsf{F}(\mathsf{a},\mathsf{b},x)$ is contained in $\Sigma(\mathsf{ren}(\mathsf{F}(x,x,x)))$. However, it is easy to see that neither $\mathsf{DG}_{\mathsf{e}}(\mathcal{P},\mathcal{R})$ nor $\mathsf{DG}_{\mathsf{nv}}(\mathcal{P},\mathcal{R})$ contain an arc from $\mathsf{F}(\mathsf{a},\mathsf{b},x) \to \mathsf{F}(x,x,x)$ to itself because $\mathsf{tcap}(\mathcal{R}^{-1},\mathsf{F}(\mathsf{a},\mathsf{b},x)) = \mathsf{F}(\mathsf{a},\mathsf{b},y)$ does not unify with $\mathsf{F}(x,x,x)$ and no ground instance of $\mathsf{F}(x,x,x)$ is contained in $\leftarrow_{\mathsf{nv}(\mathcal{R})}^{*}(\Sigma(\mathsf{F}(\mathsf{a},\mathsf{b},x)))$.

5.3.5 Innermost Dependency Graphs

In the following we show how the ideas presented in Subsections 5.3.2 and 5.3.3 can be extended to innermost termination. Let $\mathcal{R}$ be a TRS. The *innermost relation* $\xrightarrow{\mathsf{i}}$ of $\mathcal{R}$ is defined as $s \xrightarrow{\mathsf{i}}_{\mathcal{R}} t$ if there is a rewrite rule $l \to r \in \mathcal{R}$, a position $p \in \mathcal{P}\mathsf{os}(l)$, and a substitution σ such that $s|_p = l\sigma$, $t = s[r\sigma]_p$, and for all proper subterms $u \lhd s|_p$, u is a normal from with respect to $\mathcal{R}$. We say that $\mathcal{R}$ is *innermost terminating* if it does not admit an infinite innermost rewrite sequence. Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs and $\mathcal{G} = (\mathcal{P}, \mathcal{P} \times \mathcal{P})$ a directed graph. A *minimal innermost* rewrite sequence is an infinite rewrite sequence of the form $s_1 \xrightarrow{\mathsf{i}}_{\mathcal{P}} t_1 \xrightarrow{\mathsf{i}}_{\mathcal{R}}^{*} s_2 \xrightarrow{\mathsf{i}}_{\mathcal{P}} t_2 \xrightarrow{\mathsf{i}}_{\mathcal{R}}^{*} \cdots$ such that $s_i \xrightarrow{\epsilon} t_i$ and $(\alpha_i, \alpha_{i+1}) \in \mathcal{G}$ for all $i \geqslant 1$. A DP problem $(\mathcal{P},\mathcal{R},\mathcal{G})$ is called *innermost finite* if there are no minimal innermost rewrite sequences. Similar as for full termination, a TRS $\mathcal{R}$ is innermost terminating if and only if the initial DP problem $(\mathsf{DP}(\mathcal{R}),\mathcal{R},\mathcal{G})$ with $\mathcal{G} = (\mathsf{DP}(\mathcal{R}), \mathsf{DP}(\mathcal{R}) \times \mathsf{DP}(\mathcal{R}))$ is innermost finite [28, 55]. To prove innermost finiteness of a DP problem so called *innermost DP processors* are used. In order to be employed to prove innermost termination they need to be *innermost sound*, that is, if all DP problems in a set returned by a DP processor are innermost finite then the initial DP problem is innermost finite. In addition, to ensure that a DP processor can be used to prove innermost nontermination it must be *innermost complete* which means that if one of the DP problems returned by the DP processor is not innermost finite then the original DP problem is not innermost finite.

Definition 5.55. Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. The *innermost dependency graph processor* is defined as

$$(\mathcal{P},\mathcal{R},\mathcal{G}) \mapsto \{(\mathcal{P},\mathcal{R},\mathcal{G} \cap \mathsf{DG}^{\mathsf{i}}(\mathcal{P},\mathcal{R}))\}$$

where $\mathsf{DG}^{\mathsf{i}}(\mathcal{P},\mathcal{R})$ is the *innermost dependency graph* of $\mathcal{P}$ and $\mathcal{R}$, which has the rules in $\mathcal{P}$ as nodes and there is an arc from $s \to t$ to $u \to v$ if and only if there exist substitutions σ and τ such that $t\sigma \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}} u\tau$ and $s\sigma$ and $u\tau$ are normal forms with respect to $\mathcal{R}$.

The following result is well-known [1, 27, 55].

Theorem 5.56. *The innermost dependency graph processor is innermost sound and innermost complete.* □

By incorporating right-hand sides of forward closures, arcs of the innermost dependency graph can sometimes be eliminated. The only complication is that innermost rewriting is not closed under substitutions. To overcome this problem we add a fresh unary function symbol besides the constant # to the signature and assume that $\Sigma^{\mathsf{i}}_{\#}(\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R}))$ denotes the set of ground terms that are obtained from terms in $\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R})$ by instantiating the variables by terms built from # and this unary function symbol.

Definition 5.57. Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. The *improved innermost dependency graph* of $\mathcal{P}$ and $\mathcal{R}$, denoted by $\mathsf{IDG}^{\mathsf{i}}(\mathcal{P},\mathcal{R})$, has the rules in $\mathcal{P}$ as nodes and there is an arc from $s \to t$ to $u \to v$ if and only if there exist substitutions σ and τ such that $t\sigma \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}} u\tau$, $t\sigma \in \Sigma^{\mathsf{i}}_{\#}(\mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R}))$, and $s\sigma$ and $u\tau$ are normal forms with respect to $\mathcal{R}$.

The following results correspond to Lemma 5.42 and Theorem 5.44.

Lemma 5.58. *Let $\mathcal{P}$ and $\mathcal{R}$ be right-linear TRSs and $\alpha, \beta \in \mathcal{P}$. If $\mathcal{P} \cup \mathcal{R}$ admits a minimal innermost rewrite sequence in which infinitely many β-steps directly follow α-steps then $\mathsf{IDG}^{\mathsf{i}}(\mathcal{P},\mathcal{R})$ admits an arc from α to β.*

Proof. Assume that there is a minimal innermost rewrite sequence

$$s_1 \xrightarrow{\mathsf{i}}_{\mathcal{P}} t_1 \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}} s_2 \xrightarrow{\mathsf{i}}_{\mathcal{P}} t_2 \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}} s_3 \xrightarrow{\mathsf{i}}_{\mathcal{P}} \cdots$$

in which infinitely many β-steps directly follow α-steps. Since $\xrightarrow{\mathsf{i}} \subseteq \to$ we may assume without loss of generality that $s_1 \in \mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R})$ according to Lemma 5.40. Let $i \geqslant 1$ such that $s_i \xrightarrow{\mathsf{i}}_{\alpha} t_i \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}} s_{i+1} \xrightarrow{\mathsf{i}}_{\beta} t_{i+1}$. Because the set $\mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R})$ is closed under rewriting and hence also closed under innermost rewriting with respect to $\mathcal{P} \cup \mathcal{R}$ we know that the term t_i belongs to $\mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R})$. We have $t_i = \mathsf{rhs}(\alpha)\sigma$ and $s_{i+1} = \mathsf{lhs}(\beta)\tau$ for some substitutions σ and τ. Let $\mathcal{V}\mathsf{ar}(t_i) = \{x_1, \ldots, x_n\}$. From $t_i \in \mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R})$ we infer that $t_i\theta \in \Sigma^{\mathsf{i}}_{\#}(\mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R}))$ for the substitution θ that replaces the variable x_j by $f^j(\#)$ for all $j \in \{1, \ldots, n\}$. Here f denotes the fresh unary function symbol that has been added to the signature according to the definition of $\Sigma^{\mathsf{i}}_{\#}(\mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R}))$ to instantiate variables by terms build from # and f. Due to the facts that θ replaces each variable by a unique ground term in normal form and $t_i \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}} s_{i+1}$ we have $t_i\theta \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}} s_{i+1}\theta$ and hence $\mathsf{rhs}(\alpha)\sigma\theta \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}} \mathsf{lhs}(\beta)\tau\theta$. It remains to show that $\mathsf{lhs}(\alpha)\sigma\theta$ and $\mathsf{lhs}(\beta)\tau\theta$ are in normal form with respect to $\mathcal{R}$. For the latter term this is obviously the case since $\mathsf{lhs}(\beta)\tau = s_{i+1} \xrightarrow{\mathsf{i}}_{\beta} t_{i+1}$ and θ maps each variable in $\mathcal{V}\mathsf{ar}(s_{i+1}) \subseteq \mathcal{V}\mathsf{ar}(t_i)$ to a unique ground term in normal form. In case of $\mathsf{lhs}(\alpha)\sigma\theta$ there is also no problem because $\mathsf{lhs}(\alpha)\sigma = s_i \xrightarrow{\mathsf{i}}_{\alpha} t_i$ and θ replaces all variables in $\mathcal{V}\mathsf{ar}(s_i)$ which are also contained in $\mathcal{V}\mathsf{ar}(t_i)$ by pairwise distinct ground terms in normal form (all other variables remain unaffected). It follows that $\mathsf{IDG}^{\mathsf{i}}(\mathcal{P},\mathcal{R})$ contains an arc from α to β. $\square$

Theorem 5.59. *The* improved innermost dependency graph processor

$$(\mathcal{P},\mathcal{R},\mathcal{G}) \mapsto \begin{cases} \{(\mathcal{P},\mathcal{R},\mathcal{G} \cap \mathsf{IDG}^{\mathsf{i}}(\mathcal{P},\mathcal{R}))\} & \textit{if } \mathcal{P}\cup\mathcal{R} \textit{ is right-linear} \\ \{(\mathcal{P},\mathcal{R},\mathcal{G} \cap \mathsf{DG}^{\mathsf{i}}(\mathcal{P},\mathcal{R}))\} & \textit{otherwise} \end{cases}$$

is innermost sound and innermost complete.

Proof. Innermost soundness is an easy consequence of Lemma 5.58 and Theorem 5.56. Innermost completeness follows from the inclusions $\mathcal{G}\cap\mathsf{DG}^{\mathsf{i}}(\mathcal{P},\mathcal{R}) \subseteq \mathcal{G}$ and $\mathcal{G} \cap \mathsf{IDG}^{\mathsf{i}}(\mathcal{P},\mathcal{R}) \subseteq \mathcal{G}$. $\square$

We want to approximate innermost dependency graphs as well as improved innermost dependency graphs as discussed in Subsections 5.3.2 and 5.3.3. However there is one problem. The completion process is aimed to close a given language under full rewriting but not under innermost rewriting. Since it is in general impossible to close a tree automaton under innermost rewriting [16] we have no other choice than to approximate $\xrightarrow{\mathsf{i}}$ by $\rightarrow$. To make use of the fact that $s\sigma$ and $u\tau$ are normal forms with respect to $\mathcal{R}$, we adapt the definition of unreachable dependency pairs in two ways. First of all, to reflect that $s\sigma$ is a normal form, we restrict $\Sigma(\mathsf{ren}(t))$ and $\Sigma^{\mathsf{i}}_{\#}(\mathsf{RFC}_t(\mathcal{P}\cup\mathcal{R}))$ to the ground instances of $\mathsf{ren}(t)$ that are obtained by substituting normal forms for the variables of $\mathsf{ren}(t)$. This is possible because $s\sigma \rightarrow_{\mathcal{P}} t\sigma$ and σ is normalized as $s\sigma$ is a normal form. To be able to approximate the set of normal forms of $\mathcal{R}$ we remove all rewrite rules of $\mathcal{R}$ which are not left-linear. This step is necessary because on the one hand for non-left-linear TRSs the set of normal forms need not be regular [15] and on the other hand linearizing all non-left-linear rewrite rules would result in an under-approximation of the set $\mathsf{NF}(\mathcal{R})$. Especially the last fact is critical because ignoring it could easily result in incorrect approximations of the (improved) innermost dependency graph. In the following we write $\mathsf{lhs\text{-}linear}(\mathcal{R})$ for the TRS containing all left-linear rewrite rules of $\mathcal{R}$. Furthermore, the set $\{t\sigma \mid x\sigma \in \mathsf{NF}(\mathsf{lhs\text{-}linear}(\mathcal{R}))$ for all $x \in \mathcal{V}\mathsf{ar}(t)\}$ of normalized instances of a term t is denoted by $\Sigma_{\mathsf{NF}}(t,\mathcal{R})$. Secondly, to make use of the normal form property of $u\tau$ we demand that only instances of u that are in normal form with respect to $\mathcal{R}$ have to be unreachable from normalized instances of $\mathsf{ren}(t)$. Here, the set of normal form instances of a term u is denoted by $\mathsf{NF}(u,\mathcal{R})$ and defined as $\mathsf{NF}(u,\mathcal{R}) = \mathsf{NF}(\mathsf{lhs\text{-}linear}(\mathcal{R})) \cap \Sigma(u)$.

Definition 5.60. Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs, $\alpha,\beta \in \mathcal{P}$, and L a language. We say that β is *innermost unreachable* from α with respect to L if there is a tree automaton $\mathcal{A}$ compatible with $\mathcal{R}$ and $L \cap \Sigma_{\mathsf{NF}}(\mathsf{ren}(\mathsf{rhs}(\alpha)), \mathcal{R})$ such that $\mathcal{A}$ is quasi-deterministic if $\mathcal{P}$ or $\mathcal{R}$ is non-left-linear and $\mathsf{NF}(\mathsf{lhs}(\beta), \mathcal{R}) \cap \mathcal{L}(\mathcal{A}) = \varnothing$.

Using the above notion, we are now ready to define an approximation of the (improved) innermost dependency graph based on tree automata completion.

Definition 5.61. Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. The nodes of the *c-innermost dependency graph* $\mathsf{DG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$ are the rewrite rules of $\mathcal{P}$ and there is no arc from α to β if and only if β is innermost unreachable from α with respect to $\Sigma_{\mathsf{NF}}(\mathsf{ren}(\mathsf{rhs}(\alpha)), \mathcal{R})$. The nodes of the *c-improved innermost dependency graph* $\mathsf{IDG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$ are the rewrite rules of $\mathcal{P}$ and there is no arc from α to β if and only if β is innermost unreachable from α with respect to $\Sigma_{\#}(\mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R}))$.

Note that we replaced the set $\Sigma^{\mathsf{i}}_{\#}(\mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R}))$ by $\Sigma_{\#}(\mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R}))$ in the above definition of the c-improved innermost dependency graph because $\xrightarrow{\mathsf{i}}$ is approximated by $\rightarrow$ and $\Sigma_{\mathsf{NF}}(\mathsf{ren}(\mathsf{rhs}(\alpha)), \mathcal{R})$ as well as $\mathsf{NF}(\mathsf{lhs}(\beta), \mathcal{R})$ consider only the left-linear rewrite rules of $\mathcal{R}$. So on the basis of these facts it is sufficient to replace every variable of $\mathsf{rhs}(\alpha)$ by the fresh constant $\#$.

Lemma 5.62. *Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. Then* $\mathsf{DG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P}, \mathcal{R}) \supseteq \mathsf{DG}^{\mathsf{i}}(\mathcal{P}, \mathcal{R})$ *and* $\mathsf{IDG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P}, \mathcal{R}) \supseteq \mathsf{IDG}^{\mathsf{i}}(\mathcal{P}, \mathcal{R})$.

Proof. Straightforward adaption of the proofs of Lemmata 5.37 and 5.47. □

In contrast to full termination only a few approximations of the innermost dependency graph are known [1, 28, 32]. The most powerful one is a variant of the estimated dependency graph defined in Subsection 5.3.4. In the remainder of this subsection we compare $\mathsf{DG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$ and $\mathsf{IDG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$ with this approximation.

Let $\mathcal{R}$ be a set of rewrite rules and t a term. The function $\mathsf{icap}(\mathcal{R}, t)$ is defined as $\mathsf{icap}(\mathcal{R}, t) = t$ if t is a variable and $\mathsf{icap}(\mathcal{R}, t) = f(\mathsf{icap}(\mathcal{R}, t_1), \ldots, \mathsf{icap}(\mathcal{R}, t_n))$ if $t = f(t_1, \ldots, t_n)$ and $f(\mathsf{icap}(\mathcal{R}, t_1), \ldots, \mathsf{icap}(\mathcal{R}, t_n))$ does not unify with any $l \in \mathsf{lhs}(\mathcal{R})$. Otherwise $\mathsf{icap}(\mathcal{R}, t) = x$ for some fresh variable x.

Definition 5.63. Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. The nodes of the *estimated innermost dependency graph* $\mathsf{DG}^{\mathsf{i}}_{\mathsf{e}}(\mathcal{P}, \mathcal{R})$ are the rewrite rules of $\mathcal{P}$ and there is an arc from $s \to t$ to $u \to v$ if and only if there are most general unifiers σ and τ such that $\mathsf{icap}(\mathcal{R}, t)\sigma = u'\sigma$, $t\tau = \mathsf{tcap}(\mathcal{R}^{-1}, u')\tau$, and $s\sigma$, $u'\sigma$, and $s\tau$ are normal forms with respect to $\mathcal{R}$. Here u' denotes a fresh variant of u that does not have common variables with s and t.

The reason why we use the function tcap in the above definition to check whether there is a $\mathcal{R}^{-1}$ rewrite sequence from and instance of u to an instance of t is that an innermost rewrite sequence $t\sigma \xrightarrow{\mathsf{i}}^*_{\mathcal{R}} u\tau$ does not necessarily guarantee the existence of the innermost sequence $u\tau \xrightarrow{\mathsf{i}}^*_{\mathcal{R}^{-1}} t\sigma$. So if we would use icap instead of tcap, the estimated dependency graph $\mathsf{DG}^{\mathsf{i}}_{\mathsf{e}}(\mathcal{P}, \mathcal{R})$ would be incorrect. An interesting and immediate consequence of the use of tcap is that it is needless to check if $u'\tau$ is a normal form. Because tcap renames all variables of u' we have $u'\tau = u'$. So if u' is not a normal form then the same holds for $u'\sigma$, independent from the given σ. From [28, 49] the following result can be derived.

Lemma 5.64. *For TRSs $\mathcal{P}$ and $\mathcal{R}$,* $\mathsf{DG}^{\mathsf{i}}_{\mathsf{e}}(\mathcal{P},\mathcal{R}) \supseteq \mathsf{DG}(\mathcal{P},\mathcal{R})$. □

The next example demonstrates that neither $\mathsf{DG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$ nor $\mathsf{IDG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$ is subsumed by $\mathsf{DG}^{\mathsf{i}}_{\mathsf{e}}(\mathcal{P},\mathcal{R})$.

Example 5.65. Consider the TRSs $\mathcal{R}$ consisting of the rules $\mathsf{f}(x,x) \to \mathsf{f}(\mathsf{a},\mathsf{b})$ and $\mathsf{a} \to \mathsf{c}$, and $\mathcal{P}$ consisting of the dependency pair $\mathsf{F}(x,x) \to \mathsf{F}(\mathsf{a},\mathsf{b})$ of $\mathcal{R}$. First we compute $\mathsf{DG}^{\mathsf{i}}_{\mathsf{e}}(\mathcal{P},\mathcal{R})$. We have $\mathsf{icap}(\mathcal{R},\mathsf{F}(\mathsf{a},\mathsf{b})) = \mathsf{F}(u,\mathsf{b})$ and $\mathsf{tcap}(\mathcal{R}^{-1},\mathsf{F}(y,y)) = \mathsf{F}(v,w)$. Here $\mathsf{F}(y,y)$ represents the fresh variant of $\mathsf{F}(x,x)$ required by Definition 5.63. Because $\mathsf{F}(u,\mathsf{b})$ unifies with $\mathsf{F}(y,y)$ using the most general unifier $\sigma = \{y \mapsto \mathsf{b}, u \mapsto \mathsf{b}\}$, $\mathsf{F}(\mathsf{a},\mathsf{b})$ unifies with $\mathsf{F}(v,w)$ using the most general unifier $\tau = \{v \mapsto \mathsf{a}, w \mapsto \mathsf{b}\}$, and $\mathsf{F}(x,x)\sigma = \mathsf{F}(x,x)$, $\mathsf{F}(y,y)\sigma = \mathsf{F}(\mathsf{b},\mathsf{b})$, as well as $\mathsf{F}(x,x)\tau = \mathsf{F}(x,x)$ are normal forms with respect to $\mathcal{R}$, we know that $\mathsf{DG}^{\mathsf{i}}_{\mathsf{e}}(\mathcal{P},\mathcal{R})$ contains an arc from $\mathsf{F}(x,x) \to \mathsf{F}(\mathsf{a},\mathsf{b})$ to itself. The graphs $\mathsf{DG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$ and $\mathsf{IDG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$ coincide and do not contain any arcs because in both cases $\mathsf{F}(x,x) \to \mathsf{F}(\mathsf{a},\mathsf{b})$ is innermost unreachable from itself. This is certified by the tree automaton $\mathcal{A}$ consisting of the final state 3 and the following transitions:

$$\mathsf{a} \to 1 \qquad \mathsf{b} \to 2 \qquad \mathsf{c} \to 1 \qquad \mathsf{F}(1,2) \to 3$$

Note that $\mathsf{F}(\mathsf{a},\mathsf{b}) \in \mathcal{L}(\mathcal{A})$ and $\mathsf{NF}(\mathsf{F}(x,x),\mathcal{R}) \cap \mathcal{L}(\mathcal{A}) = \varnothing$. Additionally we have $\mathcal{L}(\mathcal{A}) = \to^*_{\mathcal{R}}(L) = \{\mathsf{F}(\mathsf{a},\mathsf{b}),\mathsf{F}(\mathsf{c},\mathsf{b})\}$ for both $L = \Sigma_{\mathsf{NF}}(\mathsf{F}(\mathsf{a},\mathsf{b}),\mathcal{R}) = \{\mathsf{F}(\mathsf{a},\mathsf{b})\}$ and $L = \Sigma_{\#}(\mathsf{RFC}_{\mathsf{F}(\mathsf{a},\mathsf{b})}(\mathcal{P} \cup \mathcal{R})) = \{\mathsf{F}(\mathsf{a},\mathsf{b}),\mathsf{F}(\mathsf{c},\mathsf{b})\}$.

The converse directions also do not hold. First we show that $\mathsf{DG}^{\mathsf{i}}_{\mathsf{e}}(\mathcal{P},\mathcal{R})$ is not subsumed by $\mathsf{DG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$.

Example 5.66. Consider the TRSs $\mathcal{R}$ and $\mathcal{P}$ of Example 5.54. Clearly, the graph $\mathsf{DG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$ admits an arc from $\mathsf{F}(\mathsf{a},\mathsf{b},x) \to \mathsf{F}(x,x,x)$ to itself because the normalized ground instance $\mathsf{F}(\mathsf{a},\mathsf{b},\mathsf{a})$ of $\mathsf{F}(\mathsf{a},\mathsf{b},x)$ is contained in the set $\Sigma_{\mathsf{NF}}(\mathsf{ren}(\mathsf{F}(x,x,x)),\mathcal{R})$ since both a and b are normal forms. (Note that we have $\mathsf{lhs\text{-}linear}(\mathcal{R}) = \mathcal{R}$ because $\mathcal{R}$ is left-linear.) In contrast, $\mathsf{DG}^{\mathsf{i}}_{\mathsf{e}}(\mathcal{P},\mathcal{R})$ does not contain such an arc because $\mathsf{icap}(\mathcal{R},\mathsf{F}(y,y,y)) = \mathsf{F}(y,y,y)$ does not unify with $\mathsf{F}(\mathsf{a},\mathsf{b},x)$. Here $\mathsf{F}(y,y,y)$ represents the fresh variant of $\mathsf{F}(x,x,x)$ required by Definition 5.63.

The next example shows that $\mathsf{IDG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$ does not subsume $\mathsf{DG}^{\mathsf{i}}_{\mathsf{e}}(\mathcal{P},\mathcal{R})$.

Example 5.67. Consider the TRSs $\mathcal{R}$ and $\mathcal{P}$ of Example 5.52. It is easy to see that $\mathsf{IDG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$ admits arcs from 3 to 1 and 2 because $\mathsf{F}(\mathsf{a},\mathsf{a})$ is an instance of $\mathsf{F}(x,x)$ which is in normal form with respect to $\mathsf{lhs\text{-}linear}(\mathcal{R})$. The rewrite sequence $\mathsf{F}(\mathsf{a},\mathsf{g}(\#,\mathsf{b})) \to_{\mathcal{R}'_{\#}} \mathsf{F}(\mathsf{a},\mathsf{b}) \to_{\mathcal{P}'_{\#}} \mathsf{F}(\mathsf{a},\mathsf{g}(\mathsf{b},\mathsf{b}))$ together with the fact that $\mathsf{F}(\mathsf{a},\mathsf{g}(\mathsf{b},\mathsf{b})) \in \mathsf{NF}(\mathsf{F}(\mathsf{a},\mathsf{g}(x,x)),\mathcal{R})$ yields an arc from 1 to 3. Likewise, $\mathsf{IDG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$ contains arcs from 1 to 1 and 2 because $\mathsf{F}(\mathsf{a},\mathsf{g}(\#,\mathsf{b})) \to_{\mathcal{P}'_{\#}} \mathsf{F}(\mathsf{a},\mathsf{a})$ and $\mathsf{F}(\mathsf{a},\mathsf{a}) \in \mathsf{NF}(\mathsf{F}(x,x),\mathcal{R})$. Finally, we also have an arc from 3 to 3 because $\mathsf{F}(\mathsf{a},\mathsf{a}) \to_{\mathcal{P}'_{\#}} \mathsf{F}(\mathsf{a},\mathsf{g}(\mathsf{a},\mathsf{b})) \to_{\mathcal{R}'_{\#}} \mathsf{F}(\mathsf{a},\mathsf{b}) \to_{\mathcal{P}'_{\#}} \mathsf{F}(\mathsf{a},\mathsf{g}(\mathsf{b},\mathsf{b}))$ and $\mathsf{F}(\mathsf{a},\mathsf{g}(\mathsf{b},\mathsf{b}))$ is an instance of $\mathsf{F}(\mathsf{a},\mathsf{g}(x,x))$ which is in normal form with respect to $\mathsf{lhs\text{-}linear}(\mathcal{R})$. Further arcs do not exist. Hence $\mathsf{IDG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$ looks as follows:

$$\begin{tikzcd}
1 \arrow[loop above] \arrow[rr, bend left] \arrow[dr] & & 3 \arrow[loop above] \arrow[ll, bend left] \arrow[dl] \\
& 2 &
\end{tikzcd}$$

Compared to $\mathsf{IDG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$, $\mathsf{DG}^{\mathsf{i}}_{\mathsf{e}}(\mathcal{P},\mathcal{R})$ does not admit an arc from 3 to 3 because $\mathsf{icap}(\mathcal{R},\mathsf{F}(\mathsf{a},\mathsf{a})) = \mathsf{F}(\mathsf{a},\mathsf{a})$ and $\mathsf{F}(\mathsf{a},\mathsf{a})$ does not unify with $\mathsf{F}(\mathsf{a},\mathsf{g}(y,y))$.

5.4 Usable Rules

A widely used approach to increase the power of DP processors is to consider only those rewrite rules of $\mathcal{R}$ which are *usable* [1, 28, 29, 33]. Let $(\mathcal{P},\mathcal{R},\mathcal{G})$ be a DP problem. The set of *usable rules* is defined as

$$\mathcal{U}(\mathcal{P},\mathcal{R}) = \bigcup_{s\to t\in\mathcal{P}} \mathcal{U}(\mathcal{R},t)$$

where $\mathcal{U}(\mathcal{R},t) \subseteq \mathcal{R}$ denotes the smallest set of rules such that

- $\mathcal{U}(\mathcal{R},r) \subseteq \mathcal{U}(\mathcal{R},t)$ if $l \to r \in \mathcal{U}(\mathcal{R},t)$,
- $\mathcal{U}(\mathcal{R},u) \subseteq \mathcal{U}(\mathcal{R},t)$ if u is a subterm of t, and
- $l \to r \in \mathcal{U}(\mathcal{R},t)$ if $t = f(t_1,\ldots,t_n)$ and $f(\mathsf{tcap}(\mathcal{R},t_1),\ldots,\mathsf{tcap}(\mathcal{R},t_n))$ unifies with a fresh variant of a left-hand side $l \in \mathsf{lhs}(\mathcal{R})$.

Furthermore, in the case that $\mathcal{P}$ or $\mathcal{R}$ is duplicating we require that the rewrite rules $c(x,y) \to x$ and $c(x,y) \to y$ belong to $\mathcal{U}(\mathcal{P},\mathcal{R})$, where c is a fresh function symbol. The two projection rules ensure that $(\mathcal{P},\mathcal{U}(\mathcal{P},\mathcal{R}),\mathcal{G})$ admits an infinite rewrite sequence whenever $(\mathcal{P},\mathcal{R},\mathcal{G})$ is not finite. If we switch from full termination to innermost termination, we can improve the definition of usable rules by using the function icap instead of tcap. So, the set of *innermost usable rules* for a DP problem $(\mathcal{P},\mathcal{R},\mathcal{G})$ is defined as

$$\mathcal{U}_{\mathsf{i}}(\mathcal{P},\mathcal{R}) = \bigcup_{s\to t\in\mathcal{P}} \mathcal{U}_{\mathsf{i}}(\mathcal{R},t)$$

where $\mathcal{U}_{\mathsf{i}}(\mathcal{R},t) \subseteq \mathcal{R}$ denotes the smallest set of rules such that

- $\mathcal{U}_{\mathsf{i}}(\mathcal{R},r) \subseteq \mathcal{U}_{\mathsf{i}}(\mathcal{R},t)$ if $l \to r \in \mathcal{U}_{\mathsf{i}}(\mathcal{R},t)$,
- $\mathcal{U}_{\mathsf{i}}(\mathcal{R},u) \subseteq \mathcal{U}_{\mathsf{i}}(\mathcal{R},t)$ if u is a subterm of t, and
- $l \to r \in \mathcal{U}_{\mathsf{i}}(\mathcal{R},t)$ if $t = f(t_1,\ldots,t_n)$ and $f(\mathsf{icap}(\mathcal{R},t_1),\ldots,\mathsf{icap}(\mathcal{R},t_n))$ unifies with a fresh variant of a left-hand side $l \in \mathsf{lhs}(\mathcal{R})$.

Note that in contrast to full termination it is not necessary to add projection rules $c(x,y) \to x$ and $c(x,y) \to y$ for some fresh function symbol c because redexes are only contracted if all immediate arguments are in normal form. Let us illustrate the above definitions on a small example.

Example 5.68. Let $\mathcal{R}$ be the TRS consisting of the rewrite rules

$$\mathsf{f}(x,x) \to \mathsf{f}(\mathsf{g}(x,\mathsf{s}(x)),\mathsf{a}) \qquad \mathsf{g}(x,x) \to \mathsf{h}(x) \qquad \mathsf{h}(x) \to x \qquad \mathsf{a} \to \mathsf{b}$$

and let $\mathcal{P}$ be the TRS consisting of the rewrite rules $\mathsf{F}(x,x) \to \mathsf{F}(\mathsf{g}(x,\mathsf{s}(x)),\mathsf{a})$ and $\mathsf{F}(x,x) \to \mathsf{G}(x,\mathsf{s}(x))$. According to the definition of usable rules we have $\mathcal{U}(\mathcal{P},\mathcal{R}) = \mathcal{U}(\mathcal{R},\mathsf{F}(\mathsf{g}(x,\mathsf{s}(x)),\mathsf{a})) \cup \mathcal{U}(\mathcal{R},\mathsf{G}(x,\mathsf{s}(x)))$. Since the function symbols F, G, and s are not defined with respect to the TRS $\mathcal{R}$ it is easy to see that $\mathcal{U}(\mathcal{R},\mathsf{F}(\mathsf{g}(x,\mathsf{s}(x)),\mathsf{a})) = \mathcal{U}(\mathcal{R},\mathsf{g}(x,\mathsf{s}(x))) \cup \mathcal{U}(\mathcal{R},\mathsf{a})$ and $\mathcal{U}(\mathcal{R},\mathsf{G}(x,\mathsf{s}(x))) = \varnothing$. Because $\mathsf{tcap}(\mathcal{R},x) = y$, $\mathsf{tcap}(\mathcal{R},\mathsf{s}(x)) = \mathsf{s}(z)$, and $\mathsf{tcap}(\mathcal{R},\mathsf{a}) = u$ we have

$$\mathcal{U}(\mathcal{R},\mathsf{g}(x,\mathsf{s}(x))) = \{\mathsf{g}(x,x) \to \mathsf{h}(x), \mathsf{h}(x) \to x\} \qquad \mathcal{U}(\mathcal{R},\mathsf{a}) = \{\mathsf{a} \to \mathsf{b}\}$$

and hence $\mathcal{U}(\mathcal{P},\mathcal{R}) = \{\mathsf{g}(x,x) \to \mathsf{h}(x), \mathsf{h}(x) \to x, \mathsf{a} \to \mathsf{b}\}$. Note that the rewrite rule $\mathsf{h}(x) \to x$ in $\mathcal{U}(\mathcal{R},\mathsf{g}(x,\mathsf{s}(x)))$ has been obtained from the right-hand side of the rewrite rule $\mathsf{g}(x,x) \to \mathsf{h}(x)$. If we compute the innermost usable rules of $\mathcal{P}$ and $\mathcal{R}$ we get

$$\mathcal{U}_{\mathsf{i}}(\mathcal{R},\mathsf{g}(x,\mathsf{s}(x))) = \varnothing \qquad \mathcal{U}_{\mathsf{i}}(\mathcal{R},\mathsf{a}) = \{\mathsf{a} \to \mathsf{b}\}$$

because $\mathsf{icap}(\mathcal{R},x) = x$, $\mathsf{icap}(\mathcal{R},\mathsf{s}(x)) = \mathsf{s}(x)$, and $\mathsf{icap}(\mathcal{R},\mathsf{a}) = y$. It follows that $\mathcal{U}_{\mathsf{i}}(\mathcal{P},\mathcal{R}) = \{\mathsf{a} \to \mathsf{b}\}$.

The following result is well-known [27, 28, 33, 55].

Lemma 5.69. *If a DP problem $(\mathcal{P},\mathcal{R},\mathcal{G})$ admits a minimal rewrite sequence $s_1 \xrightarrow{\epsilon}_{\alpha_1} t_1 \to_{\mathcal{R}}^* s_2 \xrightarrow{\epsilon}_{\alpha_2} t_2 \to_{\mathcal{R}}^* \cdots$ then the DP problem $(\mathcal{P},\mathcal{U}(\mathcal{P},\mathcal{R}),\mathcal{G})$ admits an infinite rewrite sequence $u_1 \xrightarrow{\epsilon}_{\alpha_i} v_1 \to_{\mathcal{U}(\mathcal{P},\mathcal{R})}^* u_2 \xrightarrow{\epsilon}_{\alpha_{i+1}} v_2 \to_{\mathcal{U}(\mathcal{P},\mathcal{R})}^* \cdots$ with $i \geqslant 1$ and $\alpha_j \in \mathcal{P}$ for all $j \geqslant 1$.* □

Since in general the transformation from $(\mathcal{P},\mathcal{R},\mathcal{G})$ to $(\mathcal{P},\mathcal{U}(\mathcal{P},\mathcal{R}),\mathcal{G})$ does not preserve the minimality of infinite rewrite sequences (the property that the terms t_1, t_2, ... are terminating in the definition on page 50) if $\mathcal{P}$ or $\mathcal{R}$ is duplicating [30], it must be guaranteed that the match-bounds processors of Theorems 5.2, 5.12, 5.14, and 5.20 as well as the graph processors of Definition 5.31 and Theorem 5.44 do not rely on the minimality of infinite rewrite sequences. In case of innermost termination such complications do not appear. The subsequent result is well-known [27, 28, 55].

Lemma 5.70. *If a DP problem $(\mathcal{P},\mathcal{R},\mathcal{G})$ admits a minimal innermost rewrite sequence $s_1 \xrightarrow{\mathsf{i}}_{\alpha_1} t_1 \xrightarrow{\mathsf{i}}_{\mathcal{R}}^* s_2 \xrightarrow{\mathsf{i}}_{\alpha_2} t_2 \xrightarrow{\mathsf{i}}_{\mathcal{R}}^* \cdots$ then the DP problem $(\mathcal{P},\mathcal{U}_{\mathsf{i}}(\mathcal{P},\mathcal{R}),\mathcal{G})$ admits a minimal innermost rewrite sequence $u_1 \xrightarrow{\mathsf{i}}_{\alpha_i} v_1 \xrightarrow{\mathsf{i}}_{\mathcal{U}_{\mathsf{i}}(\mathcal{P},\mathcal{R})}^* u_2 \xrightarrow{\mathsf{i}}_{\alpha_{i+1}} v_2 \xrightarrow{\mathsf{i}}_{\mathcal{U}_{\mathsf{i}}(\mathcal{P},\mathcal{R})}^* \cdots$ with $i \geqslant 1$ and $\alpha_j \in \mathcal{P}$ for all $j \geqslant 1$.* □

Note that the revers directions of the previous lemmata do not hold. In the remainder of this section we address how usable rules can be used to improve the DP processors presented in the preceding sections. First we focus on the match-bounds processors defined in Section 5.2. Afterwards, the DP processors of Section 5.3 are considered.

5.4.1 Match-Bounds

As indicated before, if we want to retrofit the match-bounds processors with usable rules, we have to take care of the fact that usable rules do not preserve the minimality of infinite rewrite sequences. For the match-bounds processors of Theorems 5.2 and 5.14 this behavior does not constitute any problems, since e(-raise)-bounds take all infinite rewrite sequences into account.

Corollary 5.71. *Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a DP problem and L a language. If the TRS $\mathcal{P} \cup \mathcal{U}(\mathcal{P}, \mathcal{R})$ is left-linear and e-bounded for L or e-raise-bounded for L then $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite.* $\square$

Similarly, the DP processors of Theorems 5.12 and 5.20 with $e = \mathsf{top}$ also consider all infinite rewrite sequences, as stated by Lemmata 5.8 and 5.17. For $e = \mathsf{match}$ there is also no problem since $e = \mathsf{match}$ can only be used for non-duplicating systems and it is known that usable rules can be used without restrictions for non-duplicating systems.[4]

Corollary 5.72. *Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be a DP problem, $s \to t \in \mathcal{P}$, and L a language. If $\mathcal{P} \cup \mathcal{U}(\mathcal{P}, \mathcal{R})$ is left-linear and $(\mathcal{P}, \mathcal{U}(\mathcal{P}, \mathcal{R}), \mathcal{G})$ is e-DP-bounded for $s \to t$ and L then $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite if and only if $(\mathcal{P}, \mathcal{R}, \mathcal{G}) \setminus \{s \to t\}$ is finite. Likewise, if $(\mathcal{P}, \mathcal{U}(\mathcal{P}, \mathcal{R}), \mathcal{G})$ is e-raise-DP-bounded for $s \to t$ and L then $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite if and only if $(\mathcal{P}, \mathcal{R}, \mathcal{G}) \setminus \{s \to t\}$ is finite.* $\square$

Let us illustrate the above results on an example.

Example 5.73. Consider the TRS $\mathcal{R}$ consisting of the following three rewrite rules:

$$\mathsf{f}(x, \mathsf{g}(y, \mathsf{a})) \to \mathsf{g}(\mathsf{f}(x, y), x) \qquad \mathsf{f}(x, \mathsf{a}) \to x \qquad \mathsf{g}(x, \mathsf{a}) \to x$$

The dependency pairs of $\mathcal{R}$ are $\mathsf{F}(x, \mathsf{g}(y, \mathsf{a})) \to \mathsf{G}(\mathsf{f}(x, y), x)$ and $\mathsf{F}(x, \mathsf{g}(y, \mathsf{a})) \to \mathsf{F}(x, y)$. After applying the dependency graph processor of Definition 5.31 as well as the SCC processor of Definition 5.32 to the initial DP problem induced by $\mathcal{R}$ we end up with the following DP problem: $(\{s \to t\}, \mathcal{R}, \mathcal{G})$ where $s \to t$ is the second dependency pair and $\mathcal{G} = (\{s \to t\}, \{s \to t\} \times \{s \to t\})$. To prove finiteness of this DP problem we can now use the DP processors induced by Corollaries 5.71 and 5.72. Since $s \to t$ does not cause any rules to be usable we have $\mathcal{U}(\{s \to t\}, \mathcal{R}) = \varnothing$. Because $\{s \to t\}$ is match-bounded by 1 we can conclude that $(\{s \to t\}, \mathcal{R}, \mathcal{G})$ is finite by Corollary 5.71. If we apply Corollary 5.72 instead of Corollary 5.71 we obtain finiteness of the DP problem $(\{s \to t\}, \mathcal{R}, \mathcal{G})$ because $(\{s \to t\}, \mathcal{U}(\{s \to t\}, \mathcal{R}), \mathcal{G})$ is match-DP-bounded for $s \to t$ by 1. Without using usable rules both DP processors fail. The reason is that for $(\{s \to t\}, \mathcal{R}, \mathcal{G})$ roof-bounds instead of match-bounds and top-DP-bounds instead of match-DP-bounds have to be used because $\mathcal{R}$ is duplicating. However by using roof-bounds or top-DP-bounds we do not succeed in constructing a compatible tree automaton.

[4]In [27, Example 14] and [33, Theorem 23] this has been shown for a slightly different definition of usable rules. Nevertheless, this result carries over to the present setting without any problems.

Note that the DP processors obtained from Corollaries 5.71 and 5.72 are in general not more powerful than the ones of Theorems 5.2, 5.12, 5.14, and 5.20. The reason is that by using $\mathcal{U}(\mathcal{P},\mathcal{R})$ instead of $\mathcal{R}$ it is possible that for duplicating TRSs $\mathcal{P} \cup \mathcal{R}$ the DP problem $(\mathcal{P},\mathcal{U}(\mathcal{P},\mathcal{R}),\mathcal{G})$ admits a minimal rewrite sequence whereas $(\mathcal{P},\mathcal{R},\mathcal{G})$ does not.

Example 5.74. Consider the TRS $\mathcal{R}$ consisting of the rewrite rule $\mathsf{f}(\mathsf{a},\mathsf{b},x) \to \mathsf{f}(x,x,x)$. There is one dependency pair, namely $\mathsf{F}(\mathsf{a},\mathsf{b},x) \to \mathsf{F}(x,x,x)$. By using Theorem 5.2 it can be easily checked that the DP problem $(\mathsf{DP}(\mathcal{R}),\mathcal{R},\mathcal{G})$ with $\mathcal{G} = (\mathsf{DP}(\mathcal{R}),\mathsf{DP}(\mathcal{R}) \times \mathsf{DP}(\mathcal{R}))$ is roof-bounded by 1, and hence finite. Similarly, Theorem 5.12 allows to conclude finiteness of $(\mathsf{DP}(\mathcal{R}),\mathcal{R},\mathcal{G})$ because $(\mathsf{DP}(\mathcal{R}),\mathcal{R},\mathcal{G})$ is top-DP-bounded for $\mathsf{F}(\mathsf{a},\mathsf{b},x) \to \mathsf{F}(x,x,x)$ by 1. If we combine both DP processors with usable rules, finiteness of $(\mathsf{DP}(\mathcal{R}),\mathcal{R},\mathcal{G})$ can no longer be shown since $(\mathsf{DP}(\mathcal{R}),\mathcal{U}(\mathsf{DP}(\mathcal{R}),\mathcal{R}),\mathcal{G})$ admits the following minimal cyclic rewrite sequence:

$$\begin{aligned}\mathsf{F}(\mathsf{a},\mathsf{b},\mathsf{c}(\mathsf{a},\mathsf{b})) &\xrightarrow{\epsilon}_{\mathsf{DP}(\mathcal{R})} \mathsf{F}(\mathsf{c}(\mathsf{a},\mathsf{b}),\mathsf{c}(\mathsf{a},\mathsf{b}),\mathsf{c}(\mathsf{a},\mathsf{b}))\\ &\to_{\mathcal{U}(\mathsf{DP}(\mathcal{R}),\mathcal{R})} \mathsf{F}(\mathsf{a},\mathsf{c}(\mathsf{a},\mathsf{b}),\mathsf{c}(\mathsf{a},\mathsf{b}))\\ &\to_{\mathcal{U}(\mathsf{DP}(\mathcal{R}),\mathcal{R})} \mathsf{F}(\mathsf{a},\mathsf{b},\mathsf{c}(\mathsf{a},\mathsf{b}))\end{aligned}$$

Here $\mathcal{U}(\mathsf{DP}(\mathcal{R}),\mathcal{R}) = \{\mathsf{c}(x,y) \to x, \mathsf{c}(x,y) \to y\}$.

5.4.2 Dependency Graphs

For the dependency graph processor of Definition 5.31 it is already known that it can be combined with usable rules [28].

Theorem 5.75. *Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs and $\alpha,\beta \in \mathcal{P}$. If $\mathcal{P} \cup \mathcal{R}$ admits a minimal rewrite sequence in which a β-step directly follows an α-step, then $\mathsf{DG}(\mathcal{P},\mathcal{U}(\mathcal{P},\mathcal{R}))$ admits an arc from α to β.* □

Let us illustrate the above result on an example where $\mathsf{DG}(\mathcal{P},\mathcal{U}(\mathcal{P},\mathcal{R}))$ is approximated by $\mathsf{DG_c}(\mathcal{P},\mathcal{U}(\mathcal{P},\mathcal{R}))$

Example 5.76. Consider the DP problem $(\mathcal{P},\mathcal{R},\mathcal{G})$ with $\mathcal{R}$ consisting of the rewrite rules

$$\begin{aligned}&\mathsf{p}(x,x) \to \mathsf{a} && \mathsf{g}(x,x,y,y) \to \mathsf{h}(x,x,y,y)\\ &\mathsf{p}(x,x) \to \mathsf{b} && \mathsf{h}(\mathsf{a},x,\mathsf{b},y) \to \mathsf{f}(\mathsf{p}(\mathsf{a},\mathsf{b}),x,\mathsf{p}(\mathsf{a},\mathsf{b}),y)\\ &\mathsf{p}(x,x) \to x\end{aligned}$$

$\mathcal{P}$ consisting of the single rewrite rule $\mathsf{F}(\mathsf{f}(x,x,x,x)) \to \mathsf{F}(\mathsf{g}(x,x,x,x))$, and $\mathcal{G} = (\mathcal{P},\mathcal{P} \times \mathcal{P})$. Since $\mathcal{P}$ as well as $\mathcal{R}$ are non-duplicating, $\mathcal{U}(\mathcal{P},\mathcal{R})$ consists of all rules of $\mathcal{R}$ except $\mathsf{p}(x,x) \to \mathsf{a}$, $\mathsf{p}(x,x) \to \mathsf{b}$, and $\mathsf{p}(x,x) \to x$. Starting with the initial tree automaton

$$\begin{aligned}\mathsf{a} &\to 1 & \mathsf{b} &\to 2 & & & \mathsf{F}(4) &\to 5\\ \mathsf{f}(p,q,u,v) &\to 3 & \mathsf{g}(p,q,u,v) &\to 4 & \mathsf{h}(p,q,u,v) &\to 3 & \mathsf{p}(p,q) &\to 3\end{aligned}$$

where $p, q, u, v \in \{1, 2, 3, 4\}$ and 5 is the only final state, it is straightforward to construct a quasi-deterministic tree automaton $\mathcal{A}$ that is compatible with $\mathcal{U}(\mathcal{P}, \mathcal{R})$ and $\Sigma(\mathsf{F}(\mathsf{g}(u, x, y, z)))$ such that $\Sigma(\mathsf{F}(\mathsf{f}(x, x, x, x))) \cap \mathcal{L}(\mathcal{A}) = \varnothing$. Hence $\mathsf{F}(\mathsf{f}(x, x, x, x)) \to \mathsf{F}(\mathsf{g}(x, x, x, x))$ is unreachable from itself and therefore $\mathsf{DG_c}(\mathcal{P}, \mathcal{U}(\mathcal{P}, \mathcal{R}))$ is empty. If we use $\mathcal{R}$ instead of $\mathcal{U}(\mathcal{P}, \mathcal{R})$ we cannot prove the absence of the arc from $\mathsf{F}(\mathsf{f}(x, x, x, x)) \to \mathsf{F}(\mathsf{g}(x, x, x, x))$ to itself. The reason is that in every quasi-deterministic tree automaton $\mathcal{B} = (\mathcal{F}, Q, Q_f, \Delta)$ over the signature $\mathcal{F} = \{\mathsf{a}, \mathsf{b}, \mathsf{f}, \mathsf{g}, \mathsf{h}, \mathsf{p}\}$ that is compatible with $\mathcal{R}$ and $\Sigma(\mathsf{F}(\mathsf{g}(u, x, y, z)))$ we cannot distinguish between the term $\mathsf{p}(\mathsf{p}(\mathsf{a}, \mathsf{b}), \mathsf{p}(\mathsf{a}, \mathsf{b}))$ and the terms a and b due to the rewrite rules $\mathsf{p}(x, x) \to \mathsf{a}$ and $\mathsf{p}(x, x) \to \mathsf{b}$. To illustrate this effect assume to the contrary that there is such a tree automaton. We consider the derivation

$$C[\mathsf{p}(\mathsf{p}(\mathsf{a}, \mathsf{b}), \mathsf{p}(\mathsf{a}, \mathsf{b}))] \to^*_{\Delta_\phi} C[\mathsf{p}(p, p)] \to_{\Delta_\phi} C[q] \to^*_{\Delta_\phi} q_\mathsf{p}$$

for some arbitrary context C and states $p, q \in Q$ and $q_\mathsf{p} \in Q_f$. Because $\mathcal{B}$ is compatible with $\mathcal{R}$, especially with the rules $\mathsf{p}(x, x) \to \mathsf{a}$ and $\mathsf{p}(x, x) \to \mathsf{b}$, we know that the transitions $\mathsf{a} \to q$ and $\mathsf{b} \to q$ belong to Δ. This however implies that $\phi(\mathsf{a}) \succeq_\phi q$ and $\phi(\mathsf{b}) \succeq_\phi q$. So the derivation $C[q] \to^*_{\Delta_\phi} q_\mathsf{p}$ gives rise to the sequences $C[\phi(\mathsf{a})] \to^*_{\Delta_\phi} q_\mathsf{a}$ and $C[\phi(\mathsf{b})] \to^*_{\Delta_\phi} q_\mathsf{b}$ with $q_\mathsf{a}, q_\mathsf{b} \in Q$, $q_\mathsf{a} \succeq_\phi q_\mathsf{p}$, and $q_\mathsf{b} \succeq_\phi q_\mathsf{p}$ (see Lemma 3.15). Because q_a and q_b subsume q_p and $q_\mathsf{p} \in Q_f$ it follows that $q_\mathsf{a}, q_\mathsf{b} \in Q_f$. Using this property it is easy to show that $\mathsf{DG_c}(\mathcal{P}, \mathcal{R})$ admits an arc from the rewrite rule $\mathsf{F}(\mathsf{f}(x, x, x, x)) \to \mathsf{F}(\mathsf{g}(x, x, x, x))$ to itself. Consider the term $\mathsf{F}(\mathsf{g}(t, t, t, t)) \in \mathcal{L}(\mathcal{B})$ with $t = \mathsf{p}(\mathsf{p}(\mathsf{a}, \mathsf{b}), \mathsf{p}(\mathsf{a}, \mathsf{b}))$. We have $\mathsf{F}(\mathsf{h}(t, t, t, t)) \in \mathcal{L}(\mathcal{B})$ because $\mathsf{F}(\mathsf{g}(t, t, t, t)) \to_\mathcal{R} \mathsf{F}(\mathsf{h}(t, t, t, t))$. Using the property that t is indistinguishable from the terms a and b we conclude that $\mathsf{F}(\mathsf{h}(\mathsf{a}, t, \mathsf{b}, t)) \in \mathcal{L}(\mathcal{B})$. Finally, the term $\mathsf{F}(\mathsf{f}(\mathsf{p}(\mathsf{a}, \mathsf{b}), \mathsf{p}(\mathsf{a}, \mathsf{b}), \mathsf{p}(\mathsf{a}, \mathsf{b}), \mathsf{p}(\mathsf{a}, \mathsf{b})))$ belongs to $\mathcal{L}(\mathcal{B})$ according to the following rewrite sequence:

$$\begin{aligned}
\mathsf{F}(\mathsf{h}(\mathsf{a}, t, \mathsf{b}, t)) &\to_\mathcal{R} \mathsf{F}(\mathsf{f}(\mathsf{p}(\mathsf{a}, \mathsf{b}), t, \mathsf{p}(\mathsf{a}, \mathsf{b}), t)) \\
&\to_\mathcal{R} \mathsf{F}(\mathsf{f}(\mathsf{p}(\mathsf{a}, \mathsf{b}), \mathsf{p}(\mathsf{a}, \mathsf{b}), \mathsf{p}(\mathsf{a}, \mathsf{b}), t)) \\
&\to_\mathcal{R} \mathsf{F}(\mathsf{f}(\mathsf{p}(\mathsf{a}, \mathsf{b}), \mathsf{p}(\mathsf{a}, \mathsf{b}), \mathsf{p}(\mathsf{a}, \mathsf{b}), \mathsf{p}(\mathsf{a}, \mathsf{b})))
\end{aligned}$$

Hence $\mathsf{DG_c}(\mathcal{P}, \mathcal{R})$ admits an arc from $\mathsf{F}(\mathsf{f}(x, x, x, x)) \to \mathsf{F}(\mathsf{g}(x, x, x, x))$ to itself.

Although the improved dependency graph $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ relies on the minimality of infinite rewrite sequences, we obtain a similar result as for the dependency graph $\mathsf{DG}(\mathcal{P}, \mathcal{R})$ because it can only be applied to right-linear and hence non-duplicating TRSs $\mathcal{P}$ and $\mathcal{R}$.

Corollary 5.77. *Let $\mathcal{P}$ and $\mathcal{R}$ be two right-linear TRSs and $\alpha, \beta \in \mathcal{P}$. If $\mathcal{P} \cup \mathcal{R}$ admits a minimal rewrite sequence in which infinitely many β-steps directly follow α-steps, then $\mathsf{IDG}(\mathcal{P}, \mathcal{U}(\mathcal{P}, \mathcal{R}))$ admits an arc from α to β.* $\square$

In case of the improved dependency graph processor obtained from Theorem 5.75 and Corollary 5.77 it is obvious that we can benefit from the usage of usable rules because sometimes $\mathcal{P} \cup \mathcal{U}(\mathcal{P}, \mathcal{R})$ is right-linear whereas $\mathcal{P} \cup \mathcal{R}$ is not. In such a situation we can compute $\mathsf{IDG}(\mathcal{P}, \mathcal{U}(\mathcal{P}, \mathcal{R}))$ instead of $\mathsf{DG}(\mathcal{P}, \mathcal{R})$ which might entail the removal of additional arcs. The following example illustrates this effect.

Example 5.78. Consider the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ with with $\mathcal{R}$ consisting of the rewrite rules $\mathsf{f}(x, \mathsf{s}(y)) \to \mathsf{g}(\mathsf{f}(x, \mathsf{p}(y)), x)$ and $\mathsf{p}(x) \to x$, $\mathcal{P}$ consisting of the rewrite rule $\mathsf{F}(x, \mathsf{s}(y)) \to \mathsf{F}(x, \mathsf{p}(y))$, and $\mathcal{G} = (\mathcal{P}, \mathcal{P} \times \mathcal{P})$. The usable rules of $\mathcal{P}$ and $\mathcal{R}$ are $\mathcal{U}(\mathcal{P}, \mathcal{R}) = \{\mathsf{p}(x) \to x\}$. Since $\mathcal{P}$ as well as $\mathcal{U}(\mathcal{P}, \mathcal{R})$ are right-linear we can apply Corollary 5.77 where $\mathsf{IDG}(\mathcal{P}, \mathcal{U}(\mathcal{P}, \mathcal{R}))$ is estimated by $\mathsf{IDG_c}(\mathcal{P}, \mathcal{U}(\mathcal{P}, \mathcal{R}))$. It is easy to see that $\mathsf{IDG_c}(\mathcal{P}, \mathcal{U}(\mathcal{P}, \mathcal{R}))$ is empty because $\mathsf{F}(\#, \mathsf{p}(\#))$ can never be rewritten to an instance of $\mathsf{F}(x, \mathsf{s}(y))$ via rewrite rules in $(\mathcal{P} \cup \mathcal{U}(\mathcal{P}, \mathcal{R}))_{\#}$. Hence $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite. If we use the DP processor of Theorem 5.44 we have to compute $\mathsf{DG}(\mathcal{P}, \mathcal{R})$ because $\mathcal{R}$ is not right-linear. Since $\mathsf{F}(x, \mathsf{p}(\mathsf{s}(y))) \to_{\mathcal{R}} \mathsf{F}(x, \mathsf{s}(y))$ by applying the rule $\mathsf{p}(x) \to x$ we know that any approximation of $\mathsf{DG}(\mathcal{P}, \mathcal{R})$ contains an arc from $\mathsf{F}(x, \mathsf{s}(y)) \to \mathsf{F}(x, \mathsf{p}(y))$ to itself. Hence we cannot conclude finiteness.

It is somehow clear that the DP processors obtained from Theorem 5.75 and Corollary 5.77, where $\mathsf{DG}(\mathcal{P}, \mathcal{U}(\mathcal{P}, \mathcal{R}))$ is approximated by $\mathsf{DG_c}(\mathcal{P}, \mathcal{U}(\mathcal{P}, \mathcal{R}))$ and $\mathsf{IDG}(\mathcal{P}, \mathcal{U}(\mathcal{P}, \mathcal{R}))$ is approximated by $\mathsf{IDG_c}(\mathcal{P}, \mathcal{U}(\mathcal{P}, \mathcal{R}))$, are in general not more powerful than the ones of Definition 5.31 and Theorem 5.44, estimated by $\mathsf{DG_c}(\mathcal{P}, \mathcal{R})$ and $\mathsf{IDG_c}(\mathcal{P}, \mathcal{R})$. The reason is that by using $\mathcal{U}(\mathcal{P}, \mathcal{R})$ instead of $\mathcal{R}$ it is possible that for duplicating $\mathcal{P} \cup \mathcal{R}$, the two projection rules contained in $\mathcal{U}(\mathcal{P}, \mathcal{R})$ cause some additional arcs.

Example 5.79. Consider the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ with $\mathcal{R}$ consisting of the rewrite rules

$$\mathsf{g}(x, x, y, y) \to \mathsf{h}(x, x, y, y) \qquad\qquad \mathsf{h}(\mathsf{a}, x, \mathsf{b}, y) \to \mathsf{f}(x, x, y, y)$$

$\mathcal{P}$ consisting of the single rewrite rule $\mathsf{F}(\mathsf{f}(x, x, x, x)) \to \mathsf{F}(\mathsf{g}(x, x, x, x))$, and $\mathcal{G} = (\mathcal{P}, \mathcal{P} \times \mathcal{P})$. Since $\mathcal{R}$ is duplicating, the DP processors of Definition 5.31 and Theorem 5.44 where $\mathsf{DG}(\mathcal{P}, \mathcal{R})$ is estimated by $\mathsf{DG_c}(\mathcal{P}, \mathcal{R})$ and $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ is approximated by $\mathsf{IDG_c}(\mathcal{P}, \mathcal{R})$ coincide. Starting with the initial tree automaton

$$\begin{array}{ccc} \mathsf{a} \to 1 & \mathsf{b} \to 2 & \mathsf{F}(4) \to 5 \\ \mathsf{f}(p, q, u, v) \to 3 & \mathsf{g}(p, q, u, v) \to 4 & \mathsf{h}(p, q, u, v) \to 3 \end{array}$$

where $p, q, u, v \in \{1, 2, 3, 4\}$ and 5 is the only final state, it is straightforward to construct a quasi-deterministic tree automaton $\mathcal{A}$ that is compatible with $\mathcal{R}$ and $\Sigma(\mathsf{F}(\mathsf{g}(u, x, y, z)))$ such that $\Sigma(\mathsf{F}(\mathsf{f}(x, x, x, x))) \cap \mathcal{L}(\mathcal{A}) = \varnothing$. Thus $\mathsf{F}(\mathsf{f}(x, x, x, x)) \to \mathsf{F}(\mathsf{g}(x, x, x, x))$ is unreachable from itself and therefore $\mathsf{DG_c}(\mathcal{P}, \mathcal{R})$ is empty. It follows that $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite. If we take $\mathcal{U}(\mathcal{P}, \mathcal{R})$ instead of $\mathcal{R}$ we cannot prove finiteness of $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ because $\mathsf{DG_c}(\mathcal{P}, \mathcal{U}(\mathcal{P}, \mathcal{R}))$ admits an arc from $\mathsf{F}(\mathsf{f}(x, x, x, x)) \to \mathsf{F}(\mathsf{g}(x, x, x, x))$ to itself, caused by the following $\mathcal{U}(\mathcal{P}, \mathcal{R})$ rewrite sequence:

$$\begin{aligned} \mathsf{F}(\mathsf{g}(\mathsf{c}(\mathsf{a},\mathsf{b}),\mathsf{c}(\mathsf{a},\mathsf{b}),\mathsf{c}(\mathsf{a},\mathsf{b}),\mathsf{c}(\mathsf{a},\mathsf{b})))) &\to \mathsf{F}(\mathsf{h}(\mathsf{c}(\mathsf{a},\mathsf{b}),\mathsf{c}(\mathsf{a},\mathsf{b}),\mathsf{c}(\mathsf{a},\mathsf{b}),\mathsf{c}(\mathsf{a},\mathsf{b})))) \\ &\to \mathsf{F}(\mathsf{h}(\mathsf{a},\mathsf{c}(\mathsf{a},\mathsf{b}),\mathsf{c}(\mathsf{a},\mathsf{b}),\mathsf{c}(\mathsf{a},\mathsf{b})))) \\ &\to \mathsf{F}(\mathsf{h}(\mathsf{a},\mathsf{c}(\mathsf{a},\mathsf{b}),\mathsf{b},\mathsf{c}(\mathsf{a},\mathsf{b})))) \\ &\to \mathsf{F}(\mathsf{f}(\mathsf{c}(\mathsf{a},\mathsf{b}),\mathsf{c}(\mathsf{a},\mathsf{b}),\mathsf{c}(\mathsf{a},\mathsf{b}),\mathsf{c}(\mathsf{a},\mathsf{b})))) \end{aligned}$$

Here c denotes the fresh function symbol introduced by the two projection rules $\mathsf{c}(x, y) \to x$ and $\mathsf{c}(x, y) \to y$ which have been added to $\mathcal{R}$ to obtain $\mathcal{U}(\mathcal{P}, \mathcal{R})$.

In contrast to full termination it is obvious that we can combine the innermost graph processors defined in Subsection 5.3.5 with innermost usable rules because, according to Lemma 5.70, minimality of infinite rewrite sequences is preserved. Hence we obtain the following results.

Corollary 5.80. *Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs and $\alpha, \beta \in \mathcal{P}$. If $\mathcal{P} \cup \mathcal{R}$ admits a minimal innermost rewrite sequence in which a β-step directly follows an α-step, then $\mathsf{DG}^{\mathsf{i}}(\mathcal{P}, \mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R}))$ admits an arc from α to β.* □

Corollary 5.81. *Let $\mathcal{P}$ and $\mathcal{R}$ be right-linear TRSs and $\alpha, \beta \in \mathcal{P}$. If $\mathcal{P} \cup \mathcal{R}$ admits a minimal innermost rewrite sequence in which infinitely many β-steps directly follow α-steps, then $\mathsf{IDG}^{\mathsf{i}}(\mathcal{P}, \mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R}))$ admits an arc from α to β.* □

Let us illustrate the above results on two examples. The first one illustrates the positive effect of usable rules if we approximate $\mathsf{DG}^{\mathsf{i}}(\mathcal{P}, \mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R}))$ by $\mathsf{DG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P}, \mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R}))$.

Example 5.82. Let $\mathcal{R}$ be the TRS consisting of the two rules $\mathsf{g}(x, x) \to \mathsf{h}(\mathsf{a}, \mathsf{a})$ and $\mathsf{f}(\mathsf{h}(x, \mathsf{a})) \to \mathsf{f}(\mathsf{g}(x, \mathsf{f}(x)))$. There are two dependency pairs of $\mathcal{R}$:

$$1\colon \mathsf{F}(\mathsf{h}(x, \mathsf{a})) \to \mathsf{F}(\mathsf{g}(x, \mathsf{f}(x))) \qquad 2\colon \mathsf{F}(\mathsf{h}(x, \mathsf{a})) \to \mathsf{G}(x, \mathsf{f}(x))$$

Because $\mathsf{lhs\text{-}linear}(\mathcal{R})$ consists just of the rewrite rule $\mathsf{f}(\mathsf{h}(x, y)) \to \mathsf{f}(\mathsf{g}(x, \mathsf{f}(x)))$ we know that $\mathsf{F}(\mathsf{g}(\mathsf{f}(\mathsf{a}), \mathsf{f}(\mathsf{a}))) \in \Sigma_{\mathsf{NF}}(\mathsf{F}(\mathsf{g}(x, \mathsf{f}(y))), \mathcal{R})$. Together with the facts that $\mathsf{F}(\mathsf{g}(\mathsf{f}(\mathsf{a}), \mathsf{f}(\mathsf{a}))) \to_{\mathcal{R}} \mathsf{F}(\mathsf{h}(\mathsf{a}, \mathsf{a}))$ and $\mathsf{F}(\mathsf{h}(\mathsf{a}, \mathsf{a}))$ is an instance of $\mathsf{F}(\mathsf{h}(x, \mathsf{a}))$ which is in normal form with respect to $\mathsf{lhs\text{-}linear}(\mathcal{R})$ we know that $\mathsf{DG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$ admits an arc from 1 to 1 as well as from 1 to 2. In contrast the graph $\mathsf{DG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P}, \mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R}))$ is empty. We have $\mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R}) = \varnothing$ and so

$$\to^{*}_{\mathcal{U}_{\mathsf{i}}(\mathcal{P},\mathcal{R})}(\Sigma_{\mathsf{NF}}(\mathsf{F}(\mathsf{g}(x, \mathsf{f}(y))), \mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R}))) = \Sigma_{\mathsf{NF}}(\mathsf{F}(\mathsf{g}(x, \mathsf{f}(y))), \mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R}))$$

because all ground instances of the right-hand side $\mathsf{F}(\mathsf{g}(x, \mathsf{f}(y)))$, contained in the set $\Sigma_{\mathsf{NF}}(\mathsf{F}(\mathsf{g}(x, \mathsf{f}(y))), \mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R}))$, are in normal form with respect to the TRS $\mathsf{lhs\text{-}linear}(\mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R}))$.

The next example shows that we can sometimes delete additional arcs of improved innermost dependency graphs if we switch from $\mathcal{R}$ to $\mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R})$.

Example 5.83. It is not difficult to see that for the TRSs $\mathcal{R}$ and $\mathcal{P}$ with $\mathcal{R}$ consisting of the rewrite rules $\mathsf{f}(\mathsf{a}, x) \to \mathsf{f}(\mathsf{g}(\mathsf{a}, \mathsf{b}), x)$ and $\mathsf{g}(x, x) \to \mathsf{a}$, and $\mathcal{P}$ consisting of the rewrite rules

$$1\colon \mathsf{F}(\mathsf{a}, x) \to \mathsf{F}(\mathsf{g}(\mathsf{a}, \mathsf{b}), x) \qquad 2\colon \mathsf{F}(\mathsf{a}, x) \to \mathsf{G}(\mathsf{a}, \mathsf{b})$$

$\mathsf{IDG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$ contains arcs from 1 to 1 and 2 because $\mathsf{F}(\mathsf{g}(\mathsf{a}, \mathsf{b}), \#) \to_{\mathcal{R}'_{\#}} \mathsf{F}(\mathsf{a}, \#)$ by applying the rewrite rule $\mathsf{g}(x, y) \to \mathsf{a}$ contained in $\mathcal{R}'_{\#}$. If we replace $\mathcal{R}$ by $\mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R})$ the situation changes completely as $\mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R}) = \varnothing$. As a result the term $\mathsf{F}(\mathsf{g}(\mathsf{a}, \mathsf{b}), \#)$ is in normal form with respect to $(\mathcal{P} \cup \mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R}))_{\#}$ and hence $\mathsf{IDG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P}, \mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R}))$ is empty.

Similar as for full termination, the innermost DP processors of Definitions 5.55 and 5.57 are in general incomparable to the ones obtained from Corollaries 5.80 and 5.81. The reason is that by using the TRS $\mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R})$ instead of $\mathcal{R}$ it is possible that rewrite rules are removed which are essential for the finiteness of $(\mathcal{P}, \mathcal{R}, \mathcal{G})$.

Example 5.84. Consider the TRS $\mathcal{R}$ consisting of the rewrite rules

$$\mathsf{f}(\mathsf{a}, x) \to \mathsf{f}(x, x) \qquad\qquad \mathsf{a} \to \mathsf{b}$$

and the TRS $\mathcal{P}$ consisting of the dependency pair $\mathsf{F}(\mathsf{a}, x) \to \mathsf{F}(x, x)$ of $\mathcal{R}$. It is easy to see that the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ where $\mathcal{G} = (\mathcal{P}, \mathcal{P} \times \mathcal{P})$ is innermost finite because the left-hand side of the rewrite rule $\mathsf{F}(\mathsf{a}, x) \to \mathsf{F}(x, x)$ is not in normal form with respect to $\mathcal{R}$. Hence it can never be applied. This behavior is also reflected by the innermost graph processors of Definitions 5.55 and 5.57, estimated by $\mathsf{DG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$ and $\mathsf{IDG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$. Since the underlying TRS $\mathcal{R}$ is duplicating both processors compute the graph $\mathsf{DG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$. Because a is not a normal form of $\mathsf{lhs\text{-}linear}(\mathcal{R})$ we conclude that $\mathsf{F}(\mathsf{a}, x)\sigma \notin \mathsf{NF}(\mathsf{F}(\mathsf{a}, x), \mathcal{R})$ for all ground substitutions σ. Hence $\mathsf{DG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$ is empty and therefore $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is innermost finite. If we use the DP processors induced by Corollaries 5.80 and 5.81, innermost finiteness of $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ can no longer be shown since $(\mathcal{P}, \mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R}), \mathcal{G})$ admits the following minimal cyclic innermost rewrite sequence:

$$\mathsf{F}(\mathsf{a}, \mathsf{a}) \xrightarrow{\mathsf{i}}_{\mathcal{P}} \mathsf{F}(\mathsf{a}, \mathsf{a})$$

Here $\mathcal{U}_{\mathsf{i}}(\mathcal{P}, \mathcal{R}) = \varnothing$. Note that a is now a normal from because the rewrite rule $\mathsf{a} \to \mathsf{b}$ has been removed.

5.5 Summary

In this chapter we showed how the match-bound technique can be integrated into the dependency pair framework. For that purpose we introduced two new enrichments which take care of the special properties of dependency pair problems. After that we illustrated how tree automata completion can be used to approximate dependency graphs. Furthermore, we showed that by using tree automata techniques together with forward closures we can sometimes remove arcs of the exact dependency graphs. Last but not least we combined all developed DP processors with usable rules.

An important open question is whether we can use the roof enrichment in connection with dependency pairs. To ensure soundness of roof(-raise)-DP-bounds, it has to be proved that no restriction of $\mathsf{roof\text{-}DP}(\mathcal{P}, s \to t, \mathcal{R})$ to a finite signature admits a minimal rewrite sequence with infinitely many $\xrightarrow{\epsilon}_{\mathsf{roof}(s \to t)}$-steps (root $\xrightarrow{\mathsf{r}}_{\mathsf{roof}(s \to t)}$-steps). We conjecture that this claim holds for arbitrary TRSs $\mathcal{P}$ and $\mathcal{R}$. A positive solution would make additional termination proofs possible.

Chapter 6

Complexity Analysis

As soon as we have established the termination of a given TRS $\mathcal{R}$, it is natural to analyze its complexity in order to determine the amount of resources that are necessary to perform computations with $\mathcal{R}$. In the area of term rewriting the length of derivations provides a suitable measurement for the complexity of rewrite systems, as proposed by Hofbauer and Lautemann [36]. The resulting notion of *derivational complexity* relates the length of the longest rewrite sequence to the size of its starting term. Thereby it is, for instance, a suitable metric for the complexity of deciding the word problem for confluent and terminating rewrite systems. If one regards a rewrite system as a program and wants to estimate the maximal number of computation steps needed to evaluate an expression to a result, then the special shape of the starting terms—a function applied to data which is in normal form—can be taken into account. Hirokawa and Moser [34] identified this special form of complexity and named it *runtime complexity*.

To show feasible upper complexity bounds, currently only a few techniques are known. Typically, termination criteria are restricted such that a polynomial complexity of the underlying TRS can be inferred. One of the most powerful methods to establish (linear) complexity bounds is the *match-bound technique* introduced in Chapter 4. Its ability to prove termination of an arbitrary regular set of ground terms makes it one of the most powerful methods that can be used to establish (linear) runtime complexity. Other techniques used to prove polynomial upper bounds are based on *polynomial interpretations* [36], suitably restricted to admit quadratic complexity bounds, *arctic matrix interpretations* [39] to conclude linear complexity bounds, or *triangular matrix interpretations* [50] which induce polynomially long derivations (the dimension of the matrices yields the degree of the polynomial). All these methods share the property that until now they have been used directly only, meaning that a single termination technique has to orient all rules in one go. However, using direct criteria exclusively is problematic due to their restricted power. In [34, 35] Hirokawa and Moser have lifted many aspects of the dependency pair framework from termination analysis into the complexity setting, resulting in the notion of *weak dependency pairs*. So for the special case of runtime complexity for the first time a modular approach has been introduced. There the modular aspect amounts to using different interpretation based criteria for parts of the dependency graph and the usable rules. However, still all rewrite rules considered must be oriented strictly in one go and only restrictive criteria may be applied for the usable rules. In the following part we present a new approach which admits a fully modular treatment. The approach is general enough that it applies to derivational as well as runtime complexity and basic enough that it allows us to combine completely different complexity criteria such as match-bounds and triangular matrix interpretations. By the modular combination of different criteria also gains in power are achieved. These gains come in two flavors. On one hand our approach allows us to obtain lower complexity bounds for several TRSs where complexity bounds have already been established before and on the other hand we found complexity bounds for systems that could not be dealt with before automatically.

The remainder of this chapter is organized as follows. First some preliminaries about complexity analysis are fixed. In Section 6.2 we formulate a modular framework for complexity analysis based on relative termination. In Section 6.3 we show how the match-bound technique can be used for complexity analysis within this setting. Our results have been implemented in the complexity prover CaT. The technical details can be inferred from Section 6.4.

Some of the results presented in this chapter appeared already in the conference paper [61]. New contributions include an improved version of relative match-bounds as well as an extended version of the complexity framework. In addition we explain in detail how match-RT-bounds can be extended to non-duplicating TRSs, resulting in the notion of match(-raise)-RT-boundedness, and how (quasi-deterministic, raise-consistent, and) (quasi-)compatible tree automata can be used to automatically check for match(-raise)-RT-boundedness.

6.1 Preliminaries

A *relative* TRS $\mathcal{R}/\mathcal{S}$ is a pair of TRSs $\mathcal{R}$ and $\mathcal{S}$ with the induced rewrite relation $\rightarrow_{\mathcal{R}/\mathcal{S}} = \rightarrow_{\mathcal{S}}^{*} \cdot \rightarrow_{\mathcal{R}} \cdot \rightarrow_{\mathcal{S}}^{*}$. A relative TRS $\mathcal{R}/\mathcal{S}$ is called linear (left-linear, duplicating) if $\mathcal{R} \cup \mathcal{S}$ is linear (left-linear, duplicating respectively). We say that $\mathcal{R}/\mathcal{S}$ is terminating if $\rightarrow_{\mathcal{R}/\mathcal{S}}$ is well founded. In the sequel we will sometimes identify a TRS $\mathcal{R}$ with the relative TRS $\mathcal{R}/\varnothing$. Let $\mathcal{F}$ be some signature and $L \subseteq \mathcal{T}(\mathcal{F})$ a language. The *derivation length* of a term t with respect to a rewrite relation $\rightarrow$ is defined as $\mathsf{dl}(t, \rightarrow) = \max\{n \geqslant 0 \mid t \rightarrow^{n} u \text{ for some } u \in \mathcal{T}(\mathcal{F}, \mathcal{V})\}$. Here $\rightarrow^{n}$ denotes the n-th iterate of $\rightarrow$. The *complexity* of a rewrite relation $\rightarrow$ with respect to a language L, denoted by $\mathsf{cp}_{L}(n, \rightarrow)$, computes the maximal derivation length of all terms in L up to a given size and is defined as $\mathsf{cp}_{L}(n, \rightarrow) = \max\{\mathsf{dl}(t, \rightarrow) \mid t \in L \text{ and } |t| \leqslant n\}$. Sometimes we say that a TRS $\mathcal{R}$ (relative TRS $\mathcal{R}/\mathcal{S}$) has a linear, quadratic, etc. complexity with respect to L if $\mathsf{cp}_{L}(n, \rightarrow_{\mathcal{R}})$ ($\mathsf{cp}_{L}(n, \rightarrow_{\mathcal{R}/\mathcal{S}})$) can be bounded by a linear, quadratic, etc. polynomial in n. Let $\mathcal{R}$ be a TRS over some signature $\mathcal{F}$. The *derivational complexity* of $\mathcal{R}$, abbreviated by $\mathsf{dc}(n, \mathcal{R})$ and defined as $\mathsf{dc}(n, \mathcal{R}) = \mathsf{cp}_{\mathcal{T}(\mathcal{F})}(n, \rightarrow_{\mathcal{R}})$, computes the complexity of $\rightarrow_{\mathcal{R}}$ with respect to all ground terms induced by the signature $\mathcal{F}$. In contrast, the *runtime complexity* of $\mathcal{R}$ just computes the maximal derivation length of all constructor-based terms: $\mathsf{rc}(n, \mathcal{R}) = \mathsf{cp}_{\mathcal{T}_{\mathcal{C}}(\mathcal{R})}(n, \rightarrow_{\mathcal{R}})$. Here, the set of *constructor-based terms* $\mathcal{T}_{\mathcal{C}}(\mathcal{R})$ is defined as the set of all terms $t = f(t_1, \ldots, t_n)$ such that $f \in \mathcal{F}\mathsf{un}_{\mathcal{D}}(\mathcal{R})$, and $t_i \in \mathcal{T}(\mathcal{F}\mathsf{un}_{\mathcal{C}}(\mathcal{R}))$ for all $i \in \{1, \dots, n\}$.

Let $M, N \in \mathcal{M}\mathsf{ul}(\mathbb{N})$ be multisets. The function $\mathsf{drop}_n(M)$ removes all occurrences of the number $n \in \mathbb{N}$ from M. So for all $m \in \mathbb{N}$ we have $\mathsf{drop}_n(M)(m) = 0$ if $m = n$ and $\mathsf{drop}_n(M)(m) = M(m)$ otherwise. The orderings $\succ^c_{\mathsf{mul}}$ and $\succeq^c_{\mathsf{mul}}$ are defined as $M \succ^c_{\mathsf{mul}} N$ if $\mathsf{drop}_c(M) \succ_{\mathsf{mul}} \mathsf{drop}_c(N)$ and $M \succeq^c_{\mathsf{mul}} N$ if $\mathsf{drop}_c(M) \succeq_{\mathsf{mul}} \mathsf{drop}_c(N)$. Let $\mathcal{F}$ be some signature. We extend $\succ^c_{\mathsf{mul}}$ and $\succeq^c_{\mathsf{mul}}$ to terms over the signature $\mathcal{F}_{\mathbb{N}}$ as follows: we have $s \succ^c_{\mathsf{mul}} t$ if $\mathcal{F}\mathsf{un}_{\mathcal{M}}(s) \succ^c_{\mathsf{mul}} \mathcal{F}\mathsf{un}_{\mathcal{M}}(t)$ and $s \succeq^c_{\mathsf{mul}} t$ if $\mathcal{F}\mathsf{un}_{\mathcal{M}}(s) \succeq^c_{\mathsf{mul}} \mathcal{F}\mathsf{un}_{\mathcal{M}}(t)$ for terms $s, t \in \mathcal{T}(\mathcal{F}_{\mathbb{N}}, \mathcal{V})$. Let $\sqsupset \in \{\succ_{\mathsf{mul}}, \succ^c_{\mathsf{mul}}\}$ and $\sqsupseteq \in \{\succeq_{\mathsf{mul}}, \succeq^c_{\mathsf{mul}}\}$ denote two orderings. A possibly infinite sequence of $\sqsupset$-steps is called a *chain*. We call $\sqsupset$ and $\sqsupseteq$ *compatible* if $\sqsupseteq \cdot \sqsupset \subseteq \sqsupset$ and $\sqsupset \cdot \sqsupseteq \subseteq \sqsupset$. Observe that the orderings $\succ_{\mathsf{mul}}$ and $\succeq_{\mathsf{mul}}$ as well as $\succ^c_{\mathsf{mul}}$ and $\succeq^c_{\mathsf{mul}}$ are compatible. Let $\mathcal{R}$ be a TRS. We say that $\mathcal{R}$ is *compatible* with an ordering $\sqsupset$ ($\sqsupseteq$) if $\to_{\mathcal{R}} \subseteq \sqsupset$ ($\to_{\mathcal{R}} \subseteq \sqsupseteq$).

6.2 Modular Complexity Analysis

In this section we present a modular approach which allows us to combine different techniques for estimating the complexity of TRSs. To achieve this goal we switch from full rewriting to relative rewriting. The fundamental idea for computing the complexity of a given TRS $\mathcal{R}$ is based on the following simple procedure. At first we transform $\mathcal{R}$ into the relative TRS $\mathcal{R}/\varnothing$. Then we try to bound the complexity of $\mathcal{R}/\varnothing$ by splitting $\mathcal{R}$ into smaller components $\mathcal{R}_1$ and $\mathcal{R}_2$ such that $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$. So, instead of estimating $\mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}})$ directly we want to bound it by $\mathsf{dl}(t, \to_{\mathcal{R}_1/\mathcal{S}_1}) + \mathsf{dl}(t, \to_{\mathcal{R}_2/\mathcal{S}_2})$. Here the TRS $\mathcal{S}_i$ with $i \in \{1, 2\}$ is defined as $(\mathcal{R} \cup \mathcal{S}) \setminus \mathcal{R}_i$. For each relative TRS $\mathcal{R}_i/\mathcal{S}_i$ with $i \in \{1, 2\}$ we can proceed in two directions: we can either split up $\mathcal{R}_i$ into smaller components or over-estimate $\mathsf{dl}(t, \to_{\mathcal{R}_i/\mathcal{S}_i})$ by applying some suitable method. The general idea is to choose the former option and simplify problems as much as possible. Afterwards we independently compute the complexity of each subproblem. Finally the complexity of the original system is determined by summing up all intermediate results. The next theorem states the main observation in this direction.

Theorem 6.1. *Let $\mathcal{R}/\mathcal{S}$ be a relative TRS and let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two TRSs such that $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$. Then $\mathsf{dl}(t, \to_{\mathcal{R}_1/\mathcal{S}_1}) + \mathsf{dl}(t, \to_{\mathcal{R}_2/\mathcal{S}_2}) \geqslant \mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}})$ for any terminating term t.*

Proof. Assume that $\mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}}) = n$. Then there exists a rewrite sequence of the form

$$t \to^*_{\mathcal{S}} t_1 \to_{\mathcal{R}} s_1 \to^*_{\mathcal{S}} t_2 \to_{\mathcal{R}} s_2 \to^*_{\mathcal{S}} t_3 \to_{\mathcal{R}} \cdots \to_{\mathcal{R}} s_n \to^*_{\mathcal{S}} t'$$

of length n. Next we investigate this sequence for the relative TRSs $\mathcal{R}_i/\mathcal{S}_i$ with $i \in \{1, 2\}$, where n_i is used to estimate how often rewrite rules from $\mathcal{R}_i$ have been applied in the original sequence. Fix i. If the original sequence does not contain a $\to_{\mathcal{R}_i}$-step then $t \to^*_{\mathcal{S}_i} t'$ and $n_i = 0$. In the other case the original rewrite sequence can be written as

$$t \to^*_{\mathcal{S}_i} t_{i_1} \to_{\mathcal{R}_i} s_{i_1} \to^*_{\mathcal{S}_i} t_{i_2} \to_{\mathcal{R}_i} s_{i_2} \to^*_{\mathcal{S}_i} t_{i_3} \to_{\mathcal{R}_i} \cdots \to_{\mathcal{R}_i} s_{i_{n_i}} \to^*_{\mathcal{S}_i} t'$$

where $1 \leqslant i_1 < i_2 < \cdots < i_{n_i} \leqslant n$. Apparently we have $n_1 + n_2 = n$ because each rewrite rule in $\mathcal{R}$ is either contained in $\mathcal{R}_1$ or $\mathcal{R}_2$. If $n_i = 0$ we obviously have $\mathsf{dl}(t, \rightarrow_{\mathcal{R}_i/\mathcal{S}_i}) \geqslant n_i$ and if $t \rightarrow^{n_i}_{\mathcal{R}_i/\mathcal{S}_i} t'$ with $n_i > 0$ we know that $\mathsf{dl}(t, \rightarrow_{\mathcal{R}_i/\mathcal{S}_i}) \geqslant n_i$ by the choice of the rewrite sequence. (Note that in both cases it can happen that $\mathsf{dl}(t, \rightarrow_{\mathcal{R}_i/\mathcal{S}_i}) > n_i$ because the chosen rewrite sequence need not be maximal with respect to the relative TRS $\rightarrow_{\mathcal{R}_i/\mathcal{S}_i}$, although it is maximal for $\mathcal{R}/\mathcal{S}$.) Putting things together yields

$$\mathsf{dl}(t, \rightarrow_{\mathcal{R}_1/\mathcal{S}_1}) + \mathsf{dl}(t, \rightarrow_{\mathcal{R}_2/\mathcal{S}_2}) \geqslant n_1 + n_2 = n = \mathsf{dl}(t, \rightarrow_{\mathcal{R}/\mathcal{S}})$$

which concludes the proof. □

As already indicated in the above proof, the reverse direction of Theorem 6.1 does not hold. This is illustrated by the following example.

Example 6.2. Consider the relative TRS $\mathcal{R}/\mathcal{S}$ with $\mathcal{R}$ consisting of the two rewrite rules $\mathsf{a} \rightarrow \mathsf{b}$ and $\mathsf{a} \rightarrow \mathsf{c}$ and $\mathcal{S} = \varnothing$. We have $\mathsf{a} \rightarrow_{\mathcal{R}/\mathcal{S}} \mathsf{b}$ and $\mathsf{a} \rightarrow_{\mathcal{R}/\mathcal{S}} \mathsf{c}$. Hence $\mathsf{dl}(\mathsf{a}, \rightarrow_{\mathcal{R}/\mathcal{S}}) = 1$. However, if $\mathcal{R}_1$ consists of the rewrite rule $\mathsf{a} \rightarrow \mathsf{b}$ and $\mathcal{R}_2$ of the rewrite rule $\mathsf{a} \rightarrow \mathsf{c}$, the sum of the derivation lengths $\mathsf{dl}(\mathsf{a}, \rightarrow_{\mathcal{R}_1/\mathcal{S}_1})$ and $\mathsf{dl}(\mathsf{a}, \rightarrow_{\mathcal{R}_2/\mathcal{S}_2})$ is 2.

Although for Theorem 6.1 equality cannot be established the next result states that for complexity analysis this does not matter.

Theorem 6.3. *Let $\mathcal{R}/\mathcal{S}$ be a relative TRS and let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two TRSs such that $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$. Then for any language L we have* $\mathsf{cp}_L(n, \mathcal{R}/\mathcal{S}) \in \Theta(\mathsf{cp}_L(n, \mathcal{R}_1/\mathcal{S}_1) + \mathsf{cp}_L(n, \mathcal{R}_2/\mathcal{S}_2))$.

Proof. To prove the theorem we show that there are constants M, N and M', N' such that for any term $t \in L$ that terminates with respect to $\mathcal{R}/\mathcal{S}$ the following two properties hold:

- $\mathsf{dl}(t, \rightarrow_{\mathcal{R}/\mathcal{S}}) \leqslant M \cdot (\mathsf{dl}(t, \rightarrow_{\mathcal{R}_1/\mathcal{S}_1}) + \mathsf{dl}(t, \rightarrow_{\mathcal{R}_2/\mathcal{S}_2})) + N$
- $M' \cdot \mathsf{dl}(t, \rightarrow_{\mathcal{R}/\mathcal{S}}) + N' \geqslant \mathsf{dl}(t, \rightarrow_{\mathcal{R}_1/\mathcal{S}_1}) + \mathsf{dl}(t, \rightarrow_{\mathcal{R}_2/\mathcal{S}_2})$

The result then follows immediately from this. Theorem 6.1 shows the first property with $M = 1$ and $N = 0$. To prove the second property, let $i \in \{1, 2\}$ and

$$t \rightarrow_{\mathcal{R}_i/\mathcal{S}_i} t_1 \rightarrow_{\mathcal{R}_i/\mathcal{S}_i} \cdots \rightarrow_{\mathcal{R}_i/\mathcal{S}_i} t_n$$

be a terminating $\mathcal{R}_i/\mathcal{S}_i$ rewrite sequence of length n. Because $t \rightarrow_{\mathcal{R}_i/\mathcal{S}_i} t_n$ implies $t \rightarrow^+_{\mathcal{R}/\mathcal{S}} t_n$ we obtain $\mathsf{dl}(t, \rightarrow_{\mathcal{R}_i/\mathcal{S}_i}) \leqslant \mathsf{dl}(t, \rightarrow_{\mathcal{R}/\mathcal{S}})$. By choosing $M' = 2$ and $N' = 0$ we obtain $\mathsf{dl}(t, \rightarrow_{\mathcal{R}_1/\mathcal{S}_1}) + \mathsf{dl}(t, \rightarrow_{\mathcal{R}_2/\mathcal{S}_2}) \leqslant M' \cdot \mathsf{dl}(t, \rightarrow_{\mathcal{R}/\mathcal{S}}) + N'$ as desired. □

Theorems 6.1 and 6.3 allow us to split a relative TRS $\mathcal{R}/\mathcal{S}$ into smaller components $\mathcal{R}_1/\mathcal{S}_1$ and $\mathcal{R}_2/\mathcal{S}_2$ such that $\mathcal{R}_1 \cup \mathcal{R}_2 = \mathcal{R}$. Afterwards the complexity of these systems can be independently evaluated to estimate the complexity of $\mathcal{R}/\mathcal{S}$. In the next section we show in detail how the match-bound technique can be suited for relative complexity analysis.

6.3 Relative Match-Bounds

By a remark in [24] we know that the complexity of a linear TRS $\mathcal{R}$ with respect to a language L is bounded by a linear polynomial whenever $\mathcal{R}$ is match-bounded for L. It is easy to extend this result to match-raise-boundedness and hence to non-duplicating TRSs.

Theorem 6.4. *Let $\mathcal{R}$ be a TRS and L a language. If $\mathcal{R}$ is linear and match-bounded or non-duplicating and match-raise-bounded for L then $\mathsf{cp}_L(n, \to_{\mathcal{R}})$ is bounded by a linear polynomial.*

Proof. Assume that $\mathcal{R}$ is match-raise-bounded for L. (Recall that for a linear TRS $\mathcal{R}$, match-boundedness coincides with match-raise-boundedness.) Then by Theorem 4.15 we know that $\mathcal{R}$ is terminating on L. Let

$$t \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} \cdots \to_{\mathcal{R}} t_{n-1} \to_{\mathcal{R}} t_n$$

be an arbitrary (terminating) rewrite sequence with $t \in L$. Using Lemma 4.13 this derivation can be lifted to a length-preserving $\xrightarrow{\mathsf{r}}_{\mathsf{match}(\mathcal{R})}$ rewrite sequence

$$t' \xrightarrow{\mathsf{r}}_{\mathsf{match}(\mathcal{R})} t'_1 \xrightarrow{\mathsf{r}}_{\mathsf{match}(\mathcal{R})} \cdots \xrightarrow{\mathsf{r}}_{\mathsf{match}(\mathcal{R})} t'_{n-1} \xrightarrow{\mathsf{r}}_{\mathsf{match}(\mathcal{R})} t'_n$$

such that $t' = \mathsf{lift}_0(t)$ and $\mathsf{base}(t'_i) = t_i$ for all $i \in \{1, \ldots, n\}$. From the proof of Lemma 4.9 we know that $\xrightarrow{\mathsf{r}}_{\mathsf{match}(\mathcal{R})} \subseteq \succ^+_{\mathsf{mul}}$. Transitivity of $\succ_{\mathsf{mul}}$ yields $t'_i \succ_{\mathsf{mul}} t'_{i+1}$ for all $i \in \{0, \ldots, n-1\}$. Here $t'_0 = t'$. Since $\mathcal{R}$ is match-raise-bounded for L, all terms in this latter sequence belong to $\mathcal{T}(\mathcal{F}_{\{0,\ldots,c\}})$ for some $c \in \mathbb{N}$. Let k be the maximal number of function symbols occurring in some right-hand side in $\mathcal{R}$. Due to a remark in [9] we know that the length of the $\succ_{\mathsf{mul}}$ chain from t' to t'_n is bounded by $\|t'\| \cdot (k+1)^c$. Since $\|t'\| = \|t\|$ and the $\succ_{\mathsf{mul}}$ chain starting at t' is at least as long as the lifted and hence original sequence, we conclude that the length of the $\mathcal{R}$ rewrite sequence starting at the term t is bounded by $\|t\| \cdot (k+1)^c$. $\square$

Based on Theorem 6.4 it is easy to use the match-bound technique to estimate the complexity of a relative TRS $\mathcal{R}/\mathcal{S}$; just check for match(-raise)-boundedness of $\mathcal{R} \cup \mathcal{S}$. This process either succeeds by proving that the combined TRS is match(-raise)-bounded, or, when $\mathcal{R} \cup \mathcal{S}$ cannot be proved to be match(-raise)-bounded, it fails. Since the construction of a (quasi-deterministic, raise-consistent, and) (quasi-)compatible tree automaton does not terminate for TRSs that are not match(-raise)-bounded, the latter situation typically does not happen. This behavior causes two serious problems. On the one hand we cannot benefit from Theorem 6.3 because whenever we split a relative TRS $\mathcal{R}/\mathcal{S}$ into smaller components $\mathcal{R}_1/\mathcal{S}_1$ and $\mathcal{R}_2/\mathcal{S}_2$ such that $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ then $\mathcal{R}_1 \cup \mathcal{S}_1$ is match(-raise)-bounded if and only if $\mathcal{R}_2 \cup \mathcal{S}_2$ is match(-raise)-bounded since both TRSs coincide. On the other hand the match-bound technique cannot cooperate with other techniques since either linear complexity of all or none of the rules in $\mathcal{R}$ is shown. In [60] this problem has been addressed by specifying an upper bound on the heights that can be introduced by rewrite rules in $\mathsf{match}(\mathcal{S})$. So one tries to find a $c \in \mathbb{N}$ such that the maximum height of function symbols occurring in terms in derivations caused by the TRS $\mathsf{match}_{c+1}(\mathcal{R}) \cup \mathsf{match}_c(\mathcal{S}) \cup \mathsf{lift}_c(\mathcal{S})$ is at most c. If such a bound can be established we know that $\mathcal{R}/\mathcal{S}$ is terminating and hence that it admits at most linear complexity. In the following we extend this approach in two directions. At first we adapt it such that, given a relative TRS $\mathcal{R}/\mathcal{S}$, it can investigate the complexity of a single rewrite rule $s \to t \in \mathcal{R}$ relative to all other rules. To this end we introduce a new enrichment $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$. Secondly, the rewrite rules in $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$ which originate from size-preserving or size-decreasing rules in $\mathcal{S}_{s\to t}$ are labeled in such a way that they do not increase the heights of the function symbols in a contracted redex. Here $\mathcal{S}_{s\to t}$ denotes the TRS $(\mathcal{R} \cup \mathcal{S}) \setminus \{s \to t\}$.

To simplify the presentation we first consider linear TRSs. The extension to non-duplicating TRSs is explained in Subsection 6.3.2.

6.3.1 RT-Bounds for Left-Linear Relative TRSs

As proposed in [60] we design the new enrichment $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$ such that rewrite rules which do not originate from $s \to t$ may introduce function symbols with height at most height c. In addition we try to keep the heights of the function symbols in a contracted redex if a size-preserving or size-decreasing rewrite rule different from $s \to t$ (after dropping all heights) is applied.

Definition 6.5. Let $\mathcal{S}$ be a TRS over a signature $\mathcal{F}$ and $c \in \mathbb{N}$. The TRS $\mathsf{match\text{-}RT}^c(\mathcal{S})$ over the signature $\mathcal{F}_{\mathbb{N}}$ consists of all rules $l' \to \mathsf{lift}_d(r)$ such that $\mathsf{base}(l') \to r \in \mathcal{S}$ and

$$d = \min\{c, \mathsf{height}(l'(\epsilon))\}$$

if $\|\mathsf{base}(l')\| \geqslant \|r\|$ and $\mathsf{lift}_{\mathsf{height}(l'(\epsilon))}(\mathsf{base}(l')) = l'$, and

$$d = \min\{c, 1 + \mathsf{height}(l'(p)) \mid p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(l')\})$$

otherwise. Given a relative TRS $\mathcal{R}/\mathcal{S}$ and a rewrite rule $s \to t \in \mathcal{R}$, the relative TRS $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$ is defined as $\mathsf{match}(s \to t)/\mathsf{match\text{-}RT}^c(\mathcal{S}_{s\to t})$. Let $d \in \mathbb{N}$. The restriction of $\mathsf{match\text{-}RT}^c(\mathcal{S})$ to the signature $\mathcal{F}_{\{0,\dots,d\}}$ is denoted by $\mathsf{match\text{-}RT}^c_d(\mathcal{S})$. Likewise the relative TRS $\mathsf{match\text{-}RT}^c_d(\mathcal{R}, s \to t, \mathcal{S})$ is defined as $\mathsf{match}_d(s \to t)/\mathsf{match\text{-}RT}^c_d(\mathcal{S}_{s\to t})$. If $c = d$ then $\mathsf{match\text{-}RT}^c_d(\mathcal{R}, s \to t, \mathcal{S})$ is abbreviated by $\mathsf{match\text{-}RT}_c(\mathcal{R}, s \to t, \mathcal{S})$ and $\mathsf{match\text{-}RT}^c_d(\mathcal{S}) = \mathsf{match\text{-}RT}_c(\mathcal{S})$.

The reason for the introduction of the requirement $\|\mathsf{base}(l')\| \geqslant \|r\|$ in the above definition is that we need it later on to prove that the new enrichment can be used to infer a linear upper bound on the complexity of $\mathcal{R}/\mathcal{S}$. Let us illustrate the above definition on an example.

Example 6.6. Consider the relative TRS $\mathcal{R}/\mathcal{S}$ with $\mathcal{R}$ consisting of the rewrite rules $\mathsf{rev}(x) \to \mathsf{rev}'(x, \mathsf{nil})$ and $\mathsf{rev}'(\mathsf{nil}, y) \to y$, and $\mathcal{S}$ consisting of the single rule $\mathsf{rev}'(\mathsf{cons}(x, y), z) \to \mathsf{rev}'(y, \mathsf{cons}(x, z))$. Let $s \to t$ be the first of the two rewrite rules of $\mathcal{R}$. Then $\mathsf{match\text{-}RT}^1(\mathcal{S}_{s\to t})$ contains the rules

$$\begin{array}{ll}
\mathsf{rev}'_0(\mathsf{cons}_0(x, y), z) \to \mathsf{rev}'_0(y, \mathsf{cons}_0(x, z)) & \mathsf{rev}'_0(\mathsf{nil}_0, y) \to y \\
\mathsf{rev}'_0(\mathsf{cons}_1(x, y), z) \to \mathsf{rev}'_1(y, \mathsf{cons}_1(x, z)) & \mathsf{rev}'_0(\mathsf{nil}_1, y) \to y \\
\mathsf{rev}'_2(\mathsf{cons}_1(x, y), z) \to \mathsf{rev}'_1(y, \mathsf{cons}_1(x, z)) & \cdots
\end{array}$$

and the rewrite rules

$$\begin{array}{ll} \mathsf{rev}_0(x) \to \mathsf{rev}'_1(x, \mathsf{nil}_1) & \mathsf{rev}_1(x) \to \mathsf{rev}'_2(x, \mathsf{nil}_2) \\ \mathsf{rev}_2(x) \to \mathsf{rev}'_3(x, \mathsf{nil}_3) & \cdots \end{array}$$

belong to $\mathsf{match}(s \to t)$. The union of the two infinite TRSs constitutes $\mathsf{match\text{-}RT}^1(\mathcal{R}, s \to t, \mathcal{S})$.

The general idea behind the new enrichment $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$ is to prove the complexity of the rewrite rule $s \to t$ relative to all other rules. In combination with Theorem 6.3 we can then estimate the complexity of $\mathcal{R}/\mathcal{S}$ by computing the complexity of $(\mathcal{R} \setminus \{s \to t\})/(\mathcal{S} \cup \{s \to t\})$ and combining both results. So to put it bluntly, we try to move the rewrite rule $s \to t$ from $\mathcal{R}$ to $\mathcal{S}$. To achieve this we need the property defined below.

Definition 6.7. Let $\mathcal{R}/\mathcal{S}$ be a relative TRS and $s \to t \in \mathcal{R}$. We call $\mathcal{R}/\mathcal{S}$ *match-RT-bounded* for $s \to t$ and a language L if there exists a $c \in \mathbb{N}$ such that the height of function symbols occurring in terms in $\to^*_{\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})}(\mathsf{lift}_0(L))$ is at most c.

An immediate consequence of the next lemma is that every derivation in $\mathcal{R}/\mathcal{S}$ can be lifted to a length-preserving $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$ rewrite sequence. This result is used later on to infer termination and complexity results of $\mathcal{R}/\mathcal{S}$.

Lemma 6.8. *Let $\mathcal{R}/\mathcal{S}$ be a left-linear relative TRS, $s \to t \in \mathcal{R}$, and $c \in \mathbb{N}$. If $u \to_{s \to t} v$ or $u \to_{\mathcal{S}_{s \to t}} v$ then for all terms u' with $\mathsf{base}(u') = u$ there exists a term v' such that $\mathsf{base}(v') = v$ and $u' \to_{\mathsf{match}(s \to t)} v'$ or $u' \to_{\mathsf{match\text{-}RT}^c(\mathcal{S}_{s \to t})} v'$.*

Proof. Straightforward. □

To be able to prove that a rewrite rule $s \to t$ admits a linear upper complexity bound whenever the underlying relative TRS $\mathcal{R}/\mathcal{S}$ is match-RT-bounded for $s \to t$, we rely on the fact that the rewrite rules in $\mathsf{match\text{-}RT}_c(\mathcal{R}, s \to t, \mathcal{S})$ which originate from $s \to t$ are compatible with $\succ^c_{\mathsf{mul}}$ and all remaining rewrite rules are compatible with $\succeq^c_{\mathsf{mul}}$. However there is one problem. If $s \to t$ is collapsing the rewrite rule $\mathsf{lift}_c(s) \to \mathsf{lift}_c(t)$ appears in $\mathsf{match\text{-}RT}_c(\mathcal{R}, s \to t, \mathcal{S})$ which cannot be oriented via the ordering $\succ^c_{\mathsf{mul}}$ although $\mathsf{lift}_c(s) \succ_{\mathsf{mul}} \mathsf{lift}_c(t)$. The problem is that collapsing rewrite rules do not increase the heights of function symbols in a contracted redex because the right-hand sides consist just of single variables. To avoid this problem we assume in the following that $s \to t$ corresponds to some non-collapsing rewrite rule in $\mathcal{R}$. Alternatively, one could follow the approach in [61] which can handle collapsing rewrite rules because it does not not use an upper bound c to limit the heights that can be introduced by the enriched system. However, a disadvantages of this approach is that the heights of a contracted redex are increased more often. So apart from the collapsing case the approach presented here is strictly more powerful than the approach introduced in [61].

Lemma 6.9. *Let $\mathcal{R}$ and $\mathcal{S}$ be two TRSs and $c \in \mathbb{N}$. If $\mathcal{R}$ is non-duplicating and non-collapsing and $\mathcal{S}$ is non-duplicating then the TRS $\mathsf{match}_c(\mathcal{R})$ is compatible with the ordering $\succ^c_{\mathsf{mul}}$ and $\mathsf{match\text{-}RT}_c(\mathcal{S})$ is compatible with the ordering $\succeq^c_{\mathsf{mul}}$.*

Proof. From the proof of [24, Lemma 17] we know that for a non-duplicating TRS $\mathcal{R}$ and two terms s and t such that $s \to_{\mathsf{match}_c(\mathcal{R})} t$ we have $s \succ_{\mathsf{mul}} t$. So there are multisets X and Y such that $\mathcal{F}\mathsf{un}_{\mathcal{M}}(t) = (\mathcal{F}\mathsf{un}_{\mathcal{M}}(s) \setminus X) \cup Y$, $X \neq \varnothing$, and for all $d' \in Y$ there is a $d \in X$ such that $d < d'$. Because $\mathcal{R}$ is non-collapsing we know from the definition of $\mathsf{match}_c(\mathcal{R})$ that there is a $d \in X$ such that $d < c$ and $d < d'$ for all $d' \in Y$. From this it follows that $\mathsf{drop}_c(\mathcal{F}\mathsf{un}_{\mathcal{M}}(t)) = (\mathsf{drop}_c(\mathcal{F}\mathsf{un}_{\mathcal{M}}(s)) \setminus \mathsf{drop}_c(X)) \cup \mathsf{drop}_c(Y)$, $\mathsf{drop}_c(X) \neq \varnothing$, and for all $d' \in \mathsf{drop}_c(Y)$ there is a $d \in \mathsf{drop}_c(X)$ such that $d < d'$. As an immediate consequence we have $\mathsf{drop}_c(\mathcal{F}\mathsf{un}_{\mathcal{M}}(s)) \succ_{\mathsf{mul}} \mathsf{drop}_c(\mathcal{F}\mathsf{un}_{\mathcal{M}}(t))$ and hence $s \succ^c_{\mathsf{mul}} t$. Now let s and t be two terms and $l \to r$ a rewrite rule in $\mathsf{match\text{-}RT}_c(\mathcal{S})$ such that $s \to_{l \to r} t$. According to Definition 6.5 we can assume that either $l \to r$ is some non-collapsing rewrite rule in $\mathsf{match}_c(\mathcal{S})$ or $\mathcal{F}\mathsf{un}_{\mathcal{M}}(s) \supseteq \mathcal{F}\mathsf{un}_{\mathcal{M}}(t)$. Note that if the rewrite rule $l \to r$ is collapsing then $\mathcal{F}\mathsf{un}_{\mathcal{M}}(t) = \mathcal{F}\mathsf{un}_{\mathcal{M}}(s) \setminus \mathcal{F}\mathsf{un}_{\mathcal{M}}(l)$ and hence $\mathcal{F}\mathsf{un}_{\mathcal{M}}(s) \supseteq \mathcal{F}\mathsf{un}_{\mathcal{M}}(t)$. If $\|l\| \geqslant \|r\|$ and $\mathsf{lift}_{\mathsf{height}(l(\epsilon))}(\mathsf{base}(l)) = l$ then we have $\mathcal{F}\mathsf{un}_{\mathcal{M}}(s) \supseteq \mathcal{F}\mathsf{un}_{\mathcal{M}}(t)$ either because on the one hand l consists of at least equally many function symbols as r and on the other hand the height of the function symbols in l is propagated to the function symbols in r. Now let us continue with the proof. In the first case we obtain $s \succ^c_{\mathsf{mul}} t$ as before. So by the definition of $\succeq^c_{\mathsf{mul}}$ it follows that $s \succeq^c_{\mathsf{mul}} t$. If $\mathcal{F}\mathsf{un}_{\mathcal{M}}(s) \supseteq \mathcal{F}\mathsf{un}_{\mathcal{M}}(t)$ there also is no problem because we obviously have $\mathsf{drop}_c(\mathcal{F}\mathsf{un}_{\mathcal{M}}(s)) \succeq_{\mathsf{mul}} \mathsf{drop}_c(\mathcal{F}\mathsf{un}_{\mathcal{M}}(t))$ and hence $s \succeq^c_{\mathsf{mul}} t$. □

Since the length of every $\succ^c_{\mathsf{mul}}$ chain is bounded by a function linear in the size of the starting term—if the size-increase of the terms in the chain can be bounded by a constant—we can prove that the complexity induced by the relative TRS $\{s \to t\}/\mathcal{S}_{s \to t}$ on some language L is bounded by a linear polynomial if $\mathcal{R}/\mathcal{S}$ is match-RT-bounded for $s \to t$ and L.

Theorem 6.10. *Let $\mathcal{R}/\mathcal{S}$ be a linear relative TRS, $s \to t \in \mathcal{R}$ a non-collapsing rewrite rule, and L a language. If $\mathcal{R}/\mathcal{S}$ is match-RT-bounded for $s \to t$ and L then the relative TRS $\{s \to t\}/\mathcal{S}_{s \to t}$ is terminating on L. Furthermore, $\mathsf{cp}_L(n, \to_{\{s \to t\}/\mathcal{S}_{s \to t}})$ is bounded by a linear polynomial.*

Proof. First we show that $\{s \to t\}/\mathcal{S}_{s \to t}$ is terminating on L. Assume to the contrary that there is an infinite rewrite sequence of the form

$$t_1 \to_{\{s \to t\}/\mathcal{S}_{s \to t}} t_2 \to_{\{s \to t\}/\mathcal{S}_{s \to t}} t_3 \to_{\{s \to t\}/\mathcal{S}_{s \to t}} \cdots$$

with $t_1 \in L$. Because $\mathcal{R} \cup \mathcal{S}$ is left-linear and $\mathcal{R}/\mathcal{S}$ is match-RT-bounded for $s \to t$ and L by a $c \in \mathbb{N}$, according to Lemma 6.8, the above derivation can be lifted to an infinite $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$ rewrite sequence

$$t'_1 \to_{\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})} t'_2 \to_{\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})} t'_3 \to_{\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})} \cdots$$

starting from $t'_1 = \mathsf{lift}_0(t_1)$ such that $\mathsf{base}(t'_i) = t_i$ for all $i \geqslant 1$ and the height of every function symbol occurring in a term in the lifted sequence is at most c. Hence the employed rewrite rules in the derivation emanating from t'_1 must come from $\mathsf{match\text{-}RT}_c(\mathcal{R}, s \to t, \mathcal{S})$. With help of Lemma 6.9, transitivity of $\succeq^c_{\mathsf{mul}}$, and compatibility of the orderings $\succ^c_{\mathsf{mul}}$ and $\succeq^c_{\mathsf{mul}}$ we deduce that $t'_i \succ^c_{\mathsf{mul}} t'_{i+1}$

for all $i \geqslant 1$. However, this is excluded because $<$ is well-founded on $\{0, \ldots, c\}$ and hence $\succ^c_{\mathsf{mul}}$ is well-founded on $\mathcal{T}(\mathcal{F}_{\{0,\ldots,c\}}, \mathcal{V})$.

To prove the second part of the theorem, consider an arbitrary (terminating) rewrite sequence

$$u \to_{\{s\to t\}/\mathcal{S}_{s\to t}} u_1 \to_{\{s\to t\}/\mathcal{S}_{s\to t}} \cdots \to_{\{s\to t\}/\mathcal{S}_{s\to t}} u_n$$

with $u \in L$. Similar as before this rewrite sequence can be lifted to a to a length-preserving $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$ rewrite sequence

$$u' \to_{\mathsf{match\text{-}RT}^c(\mathcal{R},s\to t,\mathcal{S})} u'_1 \to_{\mathsf{match\text{-}RT}^c(\mathcal{R},s\to t,\mathcal{S})} \cdots \to_{\mathsf{match\text{-}RT}^c(\mathcal{R},s\to t,\mathcal{S})} u'_n$$

such that $u' = \mathsf{lift}_0(u)$ and $u'_i \succ^c_{\mathsf{mul}} u'_{i+1}$ for all $i \in \{0, \ldots, n-1\}$. Here $u'_0 = u'$ and $c \in \mathbb{N}$ such that the relative TRS $\mathcal{R}/\mathcal{S}$ is match-RT-bounded for $s \to t$ and L by c. Similarly as in the proof of Theorem 6.4 we can conclude that the length of the $\{s \to t\}/\mathcal{S}_{s\to t}$ rewrite sequence starting at the term u is bounded by $\|u\| \cdot (k+1)^c$ where k is the maximal number of function symbols occurring in some right-hand side in $\mathcal{R} \cup \mathcal{S}$; just replace $\succ_{\mathsf{mul}}$ by $\succ^c_{\mathsf{mul}}$. □

We conclude this subsection with an example.

Example 6.11. The relative TRS $\mathcal{R}/\mathcal{S}$ of Example 6.6 is match-RT-bounded for $s \to t$ and $\mathcal{T}(\mathcal{F})$ by 1. Here $s \to t$ denotes the rule $\mathsf{rev}(x) \to \mathsf{rev}'(x, \mathsf{nil})$ and $\mathcal{F} = \{\mathsf{nil}, \mathsf{cons}, \mathsf{rev}, \mathsf{rev}'\}$. Due to Theorem 6.10 we can conclude that the relative TRS $\{s \to t\}/\mathcal{S}_{s\to t}$ admits at most linear complexity. In Subsection 6.3.3 it is explained in detail how match-RT-boundedness can be checked automatically.

6.3.2 Raise-RT-Bounds for Non-Left-Linear Relative TRSs

In order to be able to apply Theorem 6.10 also to a non-linear and non-duplicating relative TRS $\mathcal{R}/\mathcal{S}$, we consider the relation $\xrightarrow{\mathsf{r}}_{\mathsf{match\text{-}RT}^c(\mathcal{R},s\to t,\mathcal{S})}$ instead of $\to_{\mathsf{match\text{-}RT}^c(\mathcal{R},s\to t,\mathcal{S})}$ which uses raise-rules to deal with non-left-linear rewrite rules. Thereby the rewrite relation $\xrightarrow{\mathsf{r}}_{\mathsf{match\text{-}RT}^c(\mathcal{R},s\to t,\mathcal{S})}$ is defined as $\xrightarrow{\mathsf{r}}^*_{\mathsf{match\text{-}RT}^c(\mathcal{S}_{s\to t})} \cdot \xrightarrow{\mathsf{r}}_{\mathsf{match}(s\to t)} \cdot \xrightarrow{\mathsf{r}}^*_{\mathsf{match\text{-}RT}^c(\mathcal{S}_{s\to t})}$ where $\xrightarrow{\mathsf{r}}_{\mathsf{match\text{-}RT}^c(\mathcal{S}_{s\to t})}$ is obtained from $\xrightarrow{\mathsf{r}}_{e(\mathcal{R})}$ by replacing the TRS $e(\mathcal{R})$ with $\mathsf{match\text{-}RT}^c(\mathcal{S}_{s\to t})$ in Definition 4.10. If we would not do that it might happen that rewrite sequences in $\mathcal{R}/\mathcal{S}$ cannot be lifted to sequences in $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$.

Definition 6.12. Let $\mathcal{R}/\mathcal{S}$ be a relative TRS and $s \to t \in \mathcal{R}$. We call $\mathcal{R}/\mathcal{S}$ *match-raise-RT-bounded* for $s \to t$ and a language L if there exists a number $c \in \mathbb{N}$ such that the height of function symbols occurring in terms belonging to $\xrightarrow{\mathsf{r}}^*_{\mathsf{match\text{-}RT}^c(\mathcal{R},s\to t,\mathcal{S})}(\mathsf{lift}_0(L))$ is at most c.

Note that for left-linear relative TRSs, match-raise-RT-boundedness coincides with match-RT-boundedness. By using the relation $\xrightarrow{\mathsf{r}}_{\mathsf{match\text{-}RT}^c(\mathcal{R},s\to t,\mathcal{S})}$ every derivation induced by the relative TRS $\mathcal{R}/\mathcal{S}$ can be simulated via the rewrite rules in $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$.

Lemma 6.13. *Let $\mathcal{R}/\mathcal{S}$ be a relative TRS, $s \to t \in \mathcal{R}$, and $c \in \mathbb{N}$. If $u \to_{s\to t} v$ or $u \to_{\mathcal{S}_{s\to t}} v$ then for all terms u' with* $\mathsf{base}(u') = u$ *there exists a term v' such that* $\mathsf{base}(v') = v$ *and $u' \xrightarrow{\mathsf{r}}_{\mathsf{match}(s\to t)} v'$ or $u' \xrightarrow{\mathsf{r}}_{\mathsf{match\text{-}RT}^c(\mathcal{S}_{s\to t})} v'$.*

Proof. Straightforward. □

Before we can prove that match-raise-RT-boundedness induces a linear upper bound on the complexity of $s \to t$ we have to ensure that the raise-rules implicitly used by the relation $\xrightarrow{\mathsf{r}}_{\mathsf{match\text{-}RT}^c(\mathcal{R}, s\to t, \mathcal{S})}$ can be oriented via $\succeq^c_{\mathsf{mul}}$.

Lemma 6.14. *For any signature $\mathcal{F}$ and $c \in \mathbb{N}$ it holds that $\mathsf{raise}_c(\mathcal{F})$ is compatible with the ordering $\succeq^c_{\mathsf{mul}}$.*

Proof. Assume that there are terms s and t such that $s \to_{\mathsf{raise}_c(\mathcal{F})} t$. According to the definition of $\mathsf{raise}_c(\mathcal{F})$ we have $\mathcal{F}\mathsf{un}_{\mathcal{M}}(t) = (\mathcal{F}\mathsf{un}_{\mathcal{M}}(s) \setminus \{d\}) \cup \{d+1\}$ for some height $d < c$. Thus $s \succ^c_{\mathsf{mul}} t$ and hence $s \succeq^c_{\mathsf{mul}} t$ according to the definition of $\succeq^c_{\mathsf{mul}}$. □

Using Lemma 6.14 it is easy to extend Theorem 6.10 to non-linear and non-duplicating relative TRSs.

Theorem 6.15. *Let $\mathcal{R}/\mathcal{S}$ be a non-duplicating relative TRS, $s \to t \in \mathcal{R}$ a non-collapsing rewrite rule, and L a language. If $\mathcal{R}/\mathcal{S}$ is match-raise-RT-bounded for $s \to t$ and L then the relative TRS $\{s \to t\}/\mathcal{S}_{s\to t}$ is terminating on L. Furthermore, $\mathsf{cp}_L(n, \to_{\{s\to t\}/\mathcal{S}_{s\to t}})$ is bounded by a linear polynomial.*

Proof. First we show that $\{s \to t\}/\mathcal{S}_{s\to t}$ is terminating on L. Assume to the contrary that there is an infinite rewrite sequence of the form

$$t_1 \to_{\{s\to t\}/\mathcal{S}_{s\to t}} t_2 \to_{\{s\to t\}/\mathcal{S}_{s\to t}} t_3 \to_{\{s\to t\}/\mathcal{S}_{s\to t}} \cdots$$

with $t_1 \in L$. Let $\mathcal{R}/\mathcal{S}$ be match-raise-RT-bounded for $s \to t$ and L by a $c \in \mathbb{N}$. Lemma 6.13 yields an infinite $\xrightarrow{\mathsf{r}}_{\mathsf{match\text{-}RT}^c(\mathcal{R}, s\to t, \mathcal{S})}$ rewrite sequence

$$t'_1 \xrightarrow{\mathsf{r}}_{\mathsf{match\text{-}RT}^c(\mathcal{R}, s\to t, \mathcal{S})} t'_2 \xrightarrow{\mathsf{r}}_{\mathsf{match\text{-}RT}^c(\mathcal{R}, s\to t, \mathcal{S})} t'_3 \xrightarrow{\mathsf{r}}_{\mathsf{match\text{-}RT}^c(\mathcal{R}, s\to t, \mathcal{S})} \cdots$$

starting from $t'_1 = \mathsf{lift}_0(t_1)$ such that $\mathsf{base}(t'_i) = t_i$ for all $i \geqslant 1$. Because $\mathcal{R}/\mathcal{S}$ is match-raise-RT-bounded for $s \to t$ and L by c, the height of every function symbol occurring in a term in the lifted rewrite sequence is at most c. Hence the employed rewrite rules in the derivation emanating from t'_1 must come from $\mathsf{match\text{-}RT}_c(\mathcal{R}, s \to t, \mathcal{S})$. With help of Lemmata 6.9 and 6.14, transitivity of $\succeq^c_{\mathsf{mul}}$, and compatibility of $\succ^c_{\mathsf{mul}}$ and $\succeq^c_{\mathsf{mul}}$ we deduce that $t'_i \succ^c_{\mathsf{mul}} t'_{i+1}$ for all $i \geqslant 1$. However, this is excluded because $<$ is well-founded on $\{0, \ldots, c\}$ and hence $\succ^c_{\mathsf{mul}}$ is well-founded on $\mathcal{T}(\mathcal{F}_{\{0,\ldots,c\}}, \mathcal{V})$.

To prove the second part of the theorem, consider an arbitrary (terminating) rewrite sequence

$$u \to_{\{s\to t\}/\mathcal{S}_{s\to t}} u_1 \to_{\{s\to t\}/\mathcal{S}_{s\to t}} \cdots \to_{\{s\to t\}/\mathcal{S}_{s\to t}} u_n$$

with $u \in L$. Similar as before this rewrite sequence can be lifted to a length-preserving $\xrightarrow{\mathsf{r}}_{\mathsf{match\text{-}RT}^c(\mathcal{R}, s\to t, \mathcal{S})}$ rewrite sequence

$$u' \xrightarrow{\mathsf{r}}_{\mathsf{match\text{-}RT}^c(\mathcal{R}, s\to t, \mathcal{S})} u'_1 \xrightarrow{\mathsf{r}}_{\mathsf{match\text{-}RT}^c(\mathcal{R}, s\to t, \mathcal{S})} \cdots \xrightarrow{\mathsf{r}}_{\mathsf{match\text{-}RT}^c(\mathcal{R}, s\to t, \mathcal{S})} u'_n$$

such that $u' = \mathsf{lift}_0(u)$ and $u'_i \succ^c_{\mathsf{mul}} u'_{i+1}$ for all $i \in \{0, \ldots, n-1\}$. Here $u'_0 = u'$ and $c \in \mathbb{N}$ such that the relative TRS $\mathcal{R}/\mathcal{S}$ is match-raise-RT-bounded for $s \to t$

and L by c. Similarly as in the proof of Theorem 6.4 we can conclude that the length of the $\{s \to t\}/\mathcal{S}_{s\to t}$ rewrite sequence starting at the term u is bounded by $\|u\| \cdot (k+1)^c$ where k is the maximal number of function symbols occurring in some right-hand side in $\mathcal{R} \cup \mathcal{S}$; just replace $\succ_{\mathsf{mul}}$ by $\succ^c_{\mathsf{mul}}$. $\square$

6.3.3 Automation

To prove automatically that a relative TRS is match(-raise)-RT-bounded for some language L we use (quasi-deterministic, raise-consistent, and) compatible tree automata. Here a tree automaton $\mathcal{A}$ is said to be *compatible* with a relative TRS $\mathcal{R}/\mathcal{S}$ and a language L if $\mathcal{A}$ is compatible with $\mathcal{R} \cup \mathcal{S}$ and L.

Lemma 6.16. *Let $\mathcal{R}/\mathcal{S}$ be a left-linear relative TRS, $s \to t \in \mathcal{R}$ a rewrite rule, L a language, and $c \in \mathbb{N}$. Let $\mathcal{A}$ be a tree automaton. If $\mathcal{A}$ is compatible with the relative TRS $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$ and $\mathsf{lift}_0(L)$ such that the height of each function symbol occurring in transitions in $\mathcal{A}$ is at most c then $\mathcal{R}/\mathcal{S}$ is match-RT-bounded for $s \to t$ and L.*

Proof. Easy consequence of Theorem 3.3 and Definition 6.7 by using the fact that each each function symbol occurring in transitions in $\mathcal{A}$ is at most c. $\square$

In case of non-left-linear TRSs we obtain the following result.

Lemma 6.17. *Let $\mathcal{R}/\mathcal{S}$ be a relative TRS, $s \to t \in \mathcal{R}$ a rewrite rule, L a language, and $c \in \mathbb{N}$. Let $\mathcal{A}$ be a quasi-deterministic and raise-consistent tree automaton. If $\mathcal{A}$ is compatible with $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$ and $\mathsf{lift}_0(L)$ such that the height of each function symbol occurring in transitions in $\mathcal{A}$ is at most c then $\mathcal{R}/\mathcal{S}$ is match-raise-RT-bounded for $s \to t$ and L.*

Proof. Similar as the proof of Theorem 4.24 if we take $\mathcal{F}$ to be the signature of $\mathcal{R} \cup \mathcal{S}$ and replace $e(\mathcal{R})$ by $\mathsf{match}(s \to t) \cup \mathsf{match\text{-}RT}^c(\mathcal{S}_{s\to t})$. In order to conclude match-raise-RT-boundedness we additionally rely on the fact that the height of each function symbol occurring in transitions in $\mathcal{A}$ is at most c. $\square$

To prove that $\mathcal{R}/\mathcal{S}$ is match(-raise)-RT-bounded for some rewrite rule $s \to t$ and a language L we construct a (quasi-deterministic and raise-consistent) tree automaton $\mathcal{A}$ that is compatible with $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$ and $\mathsf{lift}_0(L)$ as described in Section 4.4. To guess an appropriate c we start with $c = 0$. As soon as a new transition $f_d(q_1, \ldots, q_n) \to q$ with $d > c$ is added to the constructed tree automaton, we set $c = d$ and proceed with the construction.

Example 6.18. We show that the relative TRS $\mathcal{R}/\mathcal{S}$ of Example 6.6 over the signature $\mathcal{F} = \{\mathsf{nil}, \mathsf{cons}, \mathsf{rev}, \mathsf{rev}'\}$ is match-RT-bounded for $\mathsf{rev}(x) \to \mathsf{rev}'(x, \mathsf{nil})$ by constructing a compatible tree automaton. Let $s \to t$ denote the rewrite rule $\mathsf{rev}(x) \to \mathsf{rev}'(x, \mathsf{nil})$. As starting point we consider the initial tree automaton

$$\mathsf{nil}_0 \to 1 \qquad \mathsf{cons}_0(1,1) \to 1 \qquad \mathsf{rev}_0(1) \to 1 \qquad \mathsf{rev}'_0(1,1) \to 1$$

which accepts all ground terms over the enriched signature $\mathsf{lift}_0(\mathcal{F})$. Because we have $\mathsf{rev}_0(x) \to_{\mathsf{match}(s\to t)} \mathsf{rev}'_1(x, \mathsf{nil}_1)$ and $\mathsf{rev}_0(1) \to 1$, we add the transitions $\mathsf{nil}_1 \to 2$ and $\mathsf{rev}'_1(1,2) \to 1$. The compatibility violation caused by the rewrite rule $\mathsf{rev}'_1(\mathsf{nil}_0, y) \to_{\mathsf{match\text{-}RT}^1(\mathcal{S}_{s\to t})} y$ and the derivation $\mathsf{rev}'_1(\mathsf{nil}_0, 2) \to^* 1$ is solved by adding the transition $2 \to 1$. Note that we are currently using $\mathsf{match\text{-}RT}^1(\mathcal{S}_{s\to t})$ because the maximal height of a function symbol occurring in the underlying tree automaton is 1. Next we consider the compatibility violation $\mathsf{rev}'_1(\mathsf{cons}_0(1,1), 2) \to^* 1$ but $\mathsf{rev}'_1(1, \mathsf{cons}_1(1,2)) \not\to^* 1$ induced by the rule $\mathsf{rev}'_1(\mathsf{cons}_0(x,y), z) \to_{\mathsf{match\text{-}RT}^1(\mathcal{S}_{s\to t})} \mathsf{rev}'_1(y, \mathsf{cons}_1(x,z))$. In order to ensure $\mathsf{rev}'_1(1, \mathsf{cons}_1(1,2)) \to^* 1$ we reuse the transition $\mathsf{rev}'_1(1,2) \to 1$ and add the new transition $\mathsf{cons}_1(1,2) \to 2$. Finally, $\mathsf{rev}'_0(\mathsf{cons}_1(1,2), 1) \to^* 1$ and $\mathsf{rev}'_0(\mathsf{cons}_1(x,y), z) \to_{\mathsf{match\text{-}RT}^1(\mathcal{S}_{s\to t})} \mathsf{rev}'_1(y, \mathsf{cons}_1(x,z))$ give rise to the transition $\mathsf{cons}_1(1,1) \to 2$. After this step, the obtained tree automaton is compatible with $\mathsf{match\text{-}RT}^1(\mathcal{R}, s \to t, \mathcal{S})$. Hence $\mathcal{R}/\mathcal{S}$ is match-RT-bounded for $s \to t$ by 1. Due to Theorem 6.10 we can conclude that $\{s \to t\}/\mathcal{S}_{s\to t}$ admits a linear complexity. To compute the complexity of $\mathcal{R}/\mathcal{S}$ we can proceed by using Theorem 6.3. Since we have already established the complexity of $\{s \to t\}/\mathcal{S}_{s\to t}$ we just have to compute the complexity of the relative TRS $\{s' \to t'\}/\mathcal{S}_{s'\to t'}$ with $s' \to t' = \mathsf{rev}'(\mathsf{nil}, y) \to y$. We remark that the ordinary match-bound technique fails on $\mathcal{R}/\mathcal{S}$ because $\mathcal{R} \cup \mathcal{S}$ induces a quadratic complexity:

$$\begin{aligned}\mathsf{rev}^n(x)\sigma^m &\to \mathsf{rev}^{n-1}(\mathsf{rev}'(x, \mathsf{nil}))\sigma^m \to^m \mathsf{rev}^{n-1}(\mathsf{rev}'(\mathsf{nil}, x))\sigma^m \\ &\to \mathsf{rev}^{n-1}(x)\sigma^m \to^{(n-1)(m+2)} x\sigma^m\end{aligned}$$

with $\sigma = \{x \mapsto \mathsf{cons}(y, x)\}$) for all $n, m \geqslant 1$.

Similar as for e(-raise)-bounds we can optimize the completion procedure by constructing a (quasi-deterministic and raise-consistent) tree-automaton $\mathcal{A}$ that is quasi-compatible with $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$ and $\mathsf{lift}_0(L)$ for some appropriate $c \in \mathbb{N}$. Here $\mathcal{A}$ is said to be *quasi-compatible* with $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$ and $\mathsf{lift}_0(L)$ if $\mathcal{A}$ is quasi-compatible with $\mathsf{match}(s \to t) \cup \mathsf{match\text{-}RT}^c(\mathcal{S}_{s\to t})$ and $\mathsf{lift}_0(L)$.

Lemma 6.19. *Let $\mathcal{R}/\mathcal{S}$ be a relative TRS, $s \to t \in \mathcal{R}$ a rewrite rule, L a language, and $c \in \mathbb{N}$. Let $\mathcal{A}$ be a tree automaton such that the height of each function symbol occurring in transitions in $\mathcal{A}$ is at most c. If $\mathcal{R}/\mathcal{S}$ is left-linear and $\mathcal{A}$ is quasi-compatible with $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$ and $\mathsf{lift}_0(L)$ then $\mathcal{R}/\mathcal{S}$ is match-RT-bounded for $s \to t$ and L. If $\mathcal{A}$ is quasi-deterministic, raise-consistent, and quasi-compatible with the relative TRS $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$ and $\mathsf{lift}_0(L)$ then $\mathcal{R}/\mathcal{S}$ is match-raise-RT-bounded for $s \to t$ and L.*

Proof. Similar to the proofs of Theorems 4.29 and 4.31; just replace the TRS $e(\mathcal{R})$ by $\mathsf{match}(s \to t) \cup \mathsf{match\text{-}RT}^c(\mathcal{S}_{s\to t})$. To conclude match(-raise)-RT-boundedness we additionally need the fact that the height of each function symbol occurring in transitions in $\mathcal{A}$ is at most c. □

6.4 The Complexity Framework

To estimate the complexity of a TRS $\mathcal{R}$, we first transform $\mathcal{R}$ into the relative TRS $\mathcal{R}/\varnothing$. (If the input is already a relative TRS this step is omitted.) Afterwards we try to apply Theorem 6.3 to estimate the complexity of $\mathcal{R}/\mathcal{S}$ by splitting $\mathcal{R}$ into TRSs $\mathcal{R}_1$ and $\mathcal{R}_2$ such that $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ and $\mathsf{dl}(t, \to_{\mathcal{R}_1/\mathcal{S}_1}) + \mathsf{dl}(t, \to_{\mathcal{R}_2/\mathcal{S}_2}) \geqslant \mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}})$. This is done by moving step by step those rewrite rules from $\mathcal{R}$ to $\mathcal{S}$ of which the complexity can be bounded. In each step a different technique can be applied. As soon as $\mathcal{R}$ is empty, we can compute the complexity of $\mathcal{R}/\mathcal{S}$ by summing up all intermediate results. In the following we describe the presented approach more formally and refer to it as the *complexity framework*.

Definition 6.20. A *complexity problem* (CP problem for short) is a triple $(\mathcal{R}/\mathcal{S}, L, f)$ consisting of a relative TRS $\mathcal{R}/\mathcal{S}$, a language L, and a unary function $f\colon \mathbb{N} \to \mathbb{N}$.

To operate on CP problems so called *complexity processors* are used. Similar as in the dependency pair framework we distinguish between sound and complete complexity processors. Here sound complexity processors are used to prove upper bounds on the complexity of CP problems whereas complete complexity processors are applied to derive complexity lower bounds.

Definition 6.21. A *complexity processor* (shortened by CP processor) is a function that takes a CP problem $(\mathcal{R}/\mathcal{S}, L, f)$ as input and returns a new CP problem $(\mathcal{R}'/\mathcal{S}', L', f')$ as output. A CP processor is *sound* if

$$f(n) + \mathsf{cp}_L(n, \to_{\mathcal{R}/\mathcal{S}}) \in \mathcal{O}(f'(n) + \mathsf{cp}_{L'}(n, \to_{\mathcal{R}'/\mathcal{S}'}))$$

and it is called *complete* if

$$f(n) + \mathsf{cp}_L(n, \to_{\mathcal{R}/\mathcal{S}}) \in \Omega(f'(n) + \mathsf{cp}_{L'}(n, \to_{\mathcal{R}'/\mathcal{S}'}))$$

for all CP problems $(\mathcal{R}/\mathcal{S}, L, f)$.

To compute the complexity of a relative TRS $\mathcal{R}/\mathcal{S}$ with respect to some language L, using the complexity framework, we proceed as follows. First we transform $\mathcal{R}/\mathcal{S}$ into the initial CP problem $(\mathcal{R}/\mathcal{S}, L, f)$ where $f(n) = 0$ for all $n \in \mathbb{N}$. After that we try to move rewrite rules from the $\mathcal{R}$ to $\mathcal{S}$ by applying different CP processors. As soon as we end up with a CP problem of the form $(\varnothing/\mathcal{S}', L', f')$ the procedure stops. If all CP processors that where involved in the computation of $(\varnothing/\mathcal{S}', L', f')$ are sound we know that $\mathsf{cp}_L(n, \to_{\mathcal{R}/\mathcal{S}}) \in \mathcal{O}(f'(n))$. If the applied CP processors are complete we conclude that $\mathsf{cp}_L(n, \to_{\mathcal{R}/\mathcal{S}}) \in \Omega(f'(n))$.

Theorem 6.22. *Let $\mathcal{R}/\mathcal{S}$ be a relative TRS and L a language. Let $P_1, \ldots, P_m$ be a sequence of CP problems such that $P_i = (\mathcal{R}_i/\mathcal{S}_i, L_i, f_i)$ for all $i \in \{1, \ldots, m\}$, $\mathcal{R}_1/\mathcal{S}_1 = \mathcal{R}/\mathcal{S}$, $L_1 = L$, $f_1(n) = 0$ for all $n \in \mathbb{N}$, and $\mathcal{R}_m = \varnothing$. If for all $i \in \{1, \ldots, m-1\}$, P_{i+1} has been obtained from P_i by applying some sound CP processor, then $\mathsf{cp}_L(n, \to_{\mathcal{R}/\mathcal{S}}) \in \mathcal{O}(f_m(n))$. If for all $i \in \{1, \ldots, m-1\}$, P_{i+1} has been obtained from P_i by applying some complete CP processor, then $\mathsf{cp}_L(n, \to_{\mathcal{R}/\mathcal{S}}) \in \Omega(f_m(n))$.*

Proof. First we show that $\mathsf{cp}_L(n, \to_{\mathcal{R}/\mathcal{S}}) \in \mathcal{O}(f_m(n))$ under the assumption that all applied CP processor are sound. According to Definition 6.21 we have

$$f_i(n) + \mathsf{cp}_{L_i}(n, \to_{\mathcal{R}_i/\mathcal{S}_i}) \in \mathcal{O}(f_{i+1}(n) + \mathsf{cp}_{L_{i+1}}(n, \to_{\mathcal{R}_{i+1}/\mathcal{S}_{i+1}}))$$

for all $i \in \{1, \ldots, m-1\}$ and hence

$$f_1(n) + \mathsf{cp}_{L_1}(n, \to_{\mathcal{R}_1/\mathcal{S}_1}) \in \mathcal{O}(f_m(n) + \mathsf{cp}_{L_m}(n, \to_{\mathcal{R}_m/\mathcal{S}_m}))$$

due to the transitivity of the $\mathcal{O}$-notation. Because $\mathsf{cp}_{L_m}(n, \to_{\mathcal{R}_m/\mathcal{S}_m}) = 0$ and $f_1(n) = 0$ we conclude that $f_1(n) + \mathsf{cp}_{L_1}(n, \to_{\mathcal{R}_1/\mathcal{S}_1}) = \mathsf{cp}_{L_1}(n, \to_{\mathcal{R}_1/\mathcal{S}_1})$ and $f_m(n) + \mathsf{cp}_{L_m}(n, \to_{\mathcal{R}_m/\mathcal{S}_m}) = f_m(n)$ for all $n \in \mathbb{N}$. Putting things together yields $\mathsf{cp}_L(n, \to_{\mathcal{R}/\mathcal{S}}) = \mathsf{cp}_{L_1}(n, \to_{\mathcal{R}_1/\mathcal{S}_1}) \in \mathcal{O}(f_m(n))$. The proof that $\mathsf{cp}_L(n, \to_{\mathcal{R}/\mathcal{S}})$ belongs to $\Omega(f_m(n))$ if the applied CP processors are complete is similar to the one before; just replace $\mathcal{O}$ by Ω. □

From the preceding section the following CP processors can be derived. To simplify the presentation we write $(\mathcal{R}/\mathcal{S}, L, f) \setminus \mathcal{R}'$ for the CP problem obtained from $(\mathcal{R}/\mathcal{S}, L, f)$ by moving the rewrite rules in $\mathcal{R}' \subseteq \mathcal{R}$ from $\mathcal{R}$ to $\mathcal{S}$, that is, $(\mathcal{R}/\mathcal{S}, L, f) \setminus \mathcal{R}' = ((\mathcal{R} \setminus \mathcal{R}')/(\mathcal{S} \cup \mathcal{R}'), L, f)$.

Theorem 6.23. *The CP processor*

$$(\mathcal{R}/\mathcal{S}, L, f) \mapsto \begin{cases} (\mathcal{R}/\mathcal{S}, L, f') \setminus \{s \to t\} & \text{if } \mathcal{R}/\mathcal{S} \text{ is linear and match-} \\ & \text{RT-bounded for } s \to t \text{ and } L \\ (\mathcal{R}/\mathcal{S}, L, f) & \text{otherwise} \end{cases}$$

where $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ and $f'(n) = f(n) + n$ is sound.

Proof. Follows from Theorems 6.3 and 6.10. □

In case of non-linear but non-duplicating relative TRSs we can use the CP processor mentioned below.

Theorem 6.24. *The CP processor*

$$(\mathcal{R}/\mathcal{S}, L, f) \mapsto \begin{cases} (\mathcal{R}/\mathcal{S}, L, f') \setminus \{s \to t\} & \text{if } \mathcal{R}/\mathcal{S} \text{ is non-duplicating} \\ & \text{and match-raise-RT-bounded} \\ & \text{for } s \to t \text{ and } L \\ (\mathcal{R}/\mathcal{S}, L, f) & \text{otherwise} \end{cases}$$

where $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ and $f'(n) = f(n) + n$ is sound.

Proof. Follows from Theorems 6.3 and 6.15. □

Note that none of the above CP processors is complete as there are linear relative TRSs which are match(-raise)-RT-bounded and admit a constant complexity.

Example 6.25. Consider the CP problem $(\mathcal{R}/\mathcal{S}, L, f)$ with $\mathcal{R}$ consisting of the single rewrite rule $\mathsf{a} \to \mathsf{b}$, $\mathcal{S} = \varnothing$, $L = \mathcal{T}(\{\mathsf{a}, \mathsf{b}\})$, and $f(n) = 0$ for all $n \in \mathbb{N}$. It is obvious that $\mathcal{R}/\mathcal{S}$ is match(-raise)-RT-bounded for $\mathsf{a} \to \mathsf{b}$ and L by 1. Hence by applying Theorem 6.23 or 6.24 we obtain the CP problem $(\varnothing/\{\mathsf{a} \to \mathsf{b}\}, L, f')$ with $f'(n) = n$. However, $\mathsf{cp}_L(n, \to_{\mathcal{R}/\mathcal{S}}) = 1$ for all $n \in \mathbb{N}$. Hence $\mathsf{cp}_L(n, \to_{\mathcal{R}/\mathcal{S}}) \notin \Omega(f'(n))$.

6.5 Summary

In this chapter we introduced a modular approach for estimating the complexity of TRSs by considering relative rewriting. We showed how the match-bound technique can be lifted into the relative setting. To this end we extended the approach in [60] by using the enrichment $\mathsf{match\text{-}RT}^c(\mathcal{R}, s \to t, \mathcal{S})$ which keeps the heights of the function symbols in a contracted redex if a size-preserving or size-decreasing rewrite rule different from $s \to t$ is applied. Finally, to exploit all possible features of the modular approach in a possible implementation we presented the so called complexity framework. We remark that our setting allows a more fine-grained complexity analysis. For instance, while traditionally a quadratic complexity ensures that any rule is applied at most quadratically often, our approach can make different statements about single rules. Hence even if a proof attempt does not succeed completely, it may highlight the problematic rules.

Chapter 7

Experiments

The techniques described in the preceding chapters have been implemented in the automatic termination analyzer TTT2 [44] and the complexity tool CaT[1]. Both, TTT2 as well as CaT, are written in OCaml[2] and consist of about 32,000 lines of code. About 11% is used to implement the techniques based on tree automata completion. More information about TTT2 and CaT, including the strategies that have been used to gain the experimental data presented in this chapter, can be found in Appendix A.

The most important criterion for the successful construction of a (quasi-)compatible tree automaton is the strategy used to resolve (quasi-)compatibility violations. In TTT2 and CaT a path $t \to^* q$ is established by adding new states and transitions according to the following strategy, which is a variation of the one presented in Section 3.3:

1. Calculate all contexts $C[\Box, \dots, \Box]$, $D_1[\Box, \dots, \Box]$, $\dots$, $D_n[\Box, \dots, \Box]$ and terms $t_1, \dots, t_m \in \mathcal{T}(\mathcal{F} \cup Q)$ such that $t_i \to^*_\Delta q_{t_i}$ for all $i \in \{1, \dots, m\}$, $C[D_1[t_1, \dots, t_i], \dots, D_n[t_j, \dots, t_m]] = t$, and $D_1[q_{t_1}, \dots, q_{t_i}] \to^*_\Delta q_1$, $\dots$, $D_n[q_{t_j}, \dots, q_{t_m}] \to^*_\Delta q_n$ with $q_1, \dots, q_n, q_{t_1}, \dots, q_{t_m} \in Q$.
2. Choose among all possibilities one where the combined size of the contexts $D_1[\Box, \dots, \Box]$, $\dots$, $D_n[\Box, \dots, \Box]$ is minimal.
3. Add new transitions involving new states (using an injective abstraction function) to achieve $D_1[q_{t_1}, \dots, q_{t_i}] \to^* q_1$, $\dots$, $D_n[q_{t_j}, \dots, q_{t_m}] \to^* q_n$.

In case that the system under consideration is non-left-linear we have to ensure that the constructed automaton is quasi-deterministic and raise-consistent. Since quasi-determinisation is expensive, TTT2 and CaT collect and resolve all (quasi-)compatibility violations with respect to the current tree automaton before making the automaton quasi-deterministic and raise-consistent. Then new (quasi-)compatibility violations are determined and the process is repeated. To detect (quasi-)compatibility violations TTT2 and CaT use a matching algorithm which is quite similar to the one presented in Section 3.2. The main difference is that the implemented algorithm uses sharing to avoid expensive recalculations. As a result it is as efficient as the approach based on tree automata techniques.

In the remaining part of this chapter we report on the experiments we performed with TTT2 and CaT on version 7.0.2 of the Termination Problem Data

[1]`http://cl-informatik.uibk.ac.at/software/cat/`

[2]`http://caml.inria.fr/`

Table 7.1: Summary e-raise-bounds

	explicit			implicit			
	t	r	rm	t	r	rm	
# successes	7	7	9	13	12	16	using c
average time	323	323	287	884	302	3658	
# timeouts	148	148	146	142	143	139	
# successes	7	7	10	14	14	17	using qc
average time	209	220	4913	783	758	667	
# timeouts	148	148	145	141	141	138	

Base (TPDB for short).[3] All tests were conducted on a server equipped with eight dual-core AMD Opteron$^{\text{TM}}$ 885 processors running at a clock rate of 2.6 GHz and on 64 GB of system memory.[4] For all experiments we used a 60 seconds time limit. We remark that similar results have been obtained on a dual-core laptop.

7.1 Match-Bounds

In this section we present the experimental data that have been obtained with $\mathsf{T_TT_2}$ using the match-bound technique. As testbed we considered the 1370 TRSs of the TPDB that fulfill the variable condition: for each rewrite rule $l \to r \in \mathcal{R}$, l is not a variable and $\mathcal{V}\mathsf{ar}(l) \supseteq \mathcal{V}\mathsf{ar}(r)$. Our results are summarized in Tables 7.1 through 7.4. We list the number of successful termination attempts, the average system time needed to prove termination (measured in milliseconds), and the number of timeouts.

7.1.1 Raise-Bounds

In Tables 7.1 and 7.2 we deal with non-left-linear systems (155 TRSs in total) and test for e-raise-boundedness, both with the explicit approach for handling raise-rules described in the first paragraph of Subsection 4.4.2 and the implicit approach using raise-consistent tree automata. To be precise, in the former table we check if the given TRS $\mathcal{R}$ is e-raise-bounded for the set of ground terms induced by the signature of $\mathcal{R}$ whereas in the latter table we check if the given TRS $\mathcal{R}$ is e-raise-bounded for the set $\mathsf{RFC}_{\mathsf{rhs}(\mathcal{R})}(\mathcal{R})$. (The usage of right-hand sides of forward closures is indicated by RFC.) To ease readability we use the abbreviations t, r and m to indicate that we test for top-, roof-, and match-raise-boundedness. Since match-raise-bounds can only be used if the given TRS is non-duplicating, we combine match-raise-bounds with roof-raise-bounds (indicated by rm). That means that if the TRS under consideration

[3]`http://www.termination-portal.org/`

[4]Full experimental data can be found at `http://cl-informatik.uibk.ac.at/software/ttt2/experiments/completion/`.

Table 7.2: Summary e-raise-bounds for RFC

	explicit			implicit			
	t	r	rm	t	r	rm	
# successes	27	27	28	36	37	39	using c
average time	275	281	274	1611	1536	1471	
# timeouts	128	128	127	119	118	116	
# successes	27	27	28	37	38	40	using qc
average time	205	208	204	1006	971	940	
# timeouts	128	128	127	118	117	115	

is non-duplicating we test for match-raise-boundedness; duplicating TRSs are tested for roof-raise-boundedness. In the upper part of the tables we construct compatible tree automata (indicated by c) whereas in the lower part quasi-compatible tree automata (indicated by qc) are used.

The positive effect of right-hand sides of forward closures (see Theorem 4.45) is clearly visible. By constructing quasi-compatible tree automata instead of compatible tree automata we get some additional termination proofs as well as an average speed up of 1.5. Furthermore, our results confirm that match-raise-bounds are more powerful than roof-raise-bounds, which in turn are more powerful than top-raise-bounds.

7.1.2 DP-Bounds and Raise-DP-Bounds

Tables 7.3 and 7.4 show our results for e(-raise)-DP-bounds. Besides the SCC processor of Definition 5.32 and the dependency graph processor of Definition 5.31 where $\mathsf{DG}(\mathcal{P},\mathcal{R})$ is approximated by the estimated dependency graph $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$ of Definition 5.49, we use the following four DP processors:

- s represents the subterm criterion of [33].
- p is an instance of the reduction pair processor based on polynomial orderings with 0/1 coefficients [30].
- b denotes a combination of the DP processor of Theorem 5.2 for left-linear DP problems and the one of Theorem 5.14 for non-left-linear DP problems.
- d uses the DP processor of Theorem 5.12 for left-linear DP problems and the one of Theorem 5.20 for non-left-linear DP problems.

For the latter two, if the DP problem is non-duplicating we take $e = \mathsf{match}$. For duplicating problems we take $e = \mathsf{roof}$ for b and $e = \mathsf{top}$ for d. If the DP problem is non-left-linear, in both cases raise-rules are handled by the implicit approach. The usage of usable rules (see Corollaries 5.71 and 5.72) is indicated by ur.

Table 7.3: Summary e(-raise)-DP-bounds

		no ur		ur		
	sp	spb	spd	spb	spd	
# successes	455	517	557	566	593	
average time	325	322	989	319	807	using c
# timeouts	3	853	813	804	777	
# successes	455	521	557	567	598	
average time	325	377	974	315	903	using qc
# timeouts	3	849	813	803	772	

Table 7.4: Summary e(-raise)-DP-bounds for RFC

		no ur		ur		
	sp	spb	spd	spb	spd	
# successes	455	536	566	583	608	
average time	325	369	926	382	638	using c
# timeouts	3	834	804	787	762	
# successes	455	538	566	585	608	
average time	325	359	911	393	570	using qc
# timeouts	3	832	804	785	762	

An important criterion for the success of e(-raise)-DP-bounds is the choice of the rewrite rule from $\mathcal{P}$ that should be removed from the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ under consideration. To find a suitable rewrite rule, $\mathsf{T_TT_2}$ simply starts the construction of a (quasi-deterministic, raise-consistent, and) (quasi-)compatible tree automaton for each $s \to t \in \mathcal{P}$ in parallel. As soon as one of the processes terminates the procedure stops and returns the corresponding rewrite rule.

The advantage of the DP processors of Theorems 5.12 and 5.20 over the naive ones of Theorems 5.2 and 5.14 is clear, although the difference decreases when usable rules and right-hand sides of forward closures are in effect. Furthermore, by using quasi-compatible tree automata instead of compatible tree automata we obtain some additional termination proofs. For b this effect is clearly visible. In case of d, the corresponding numbers in Tables 7.3 and 7.4 are identical except for Table 7.4 when usable rules are in effect. The reason for this behavior is that the number of additional termination proofs is identical to the number of TRSs for which we do not obtain a termination certificate if quasi-compatible tree automaton are used (two TRSs for which $\mathsf{T_TT_2}$ can construct a quasi-compatible tree automaton but not a compatible tree automaton are Secret_05_TRS/teparla1 and Zantema_05/z30).

Although not visible from the data in Table 7.3, our experiments do not confirm the claim at the end of Subsection 5.4.1 that usable rules sometimes have an adverse effect. The reason for this behavior is that the other two processors (s and p) suffice to prove the termination of critical TRSs in the TPDB. If we would drop s and p then the termination of TRSs like SK90/4.50 can no longer be proved because, by computing usable rules in advance, the added projection rules cause the (quasi-)completion procedure to loop. Although our experiments do not demonstrate the negative effect of usable rules, they confirm that restricting the computation of usable rules to non-duplicating systems in order to avoid negative effects induced by the added projection rules is not a good strategy since there are duplicating TRSs such as Secret_06_TRS/divExp which can only be proved terminating with help of usable rules.

The TRSs Secret_07_TRS/1 and Secret_07_TRS/4 in the TPDB can be proved terminating by $\mathsf{T_TT_2}$ using match-raise-DP-bounds and right-hand sides of forward closures. In the 2009 edition of the international termination competition, besides $\mathsf{T_TT_2}$, only AProVE [26] could show the termination of the two TRSs.[5] Another interesting TRS is Secret_06_TRS/gen-25. In the 2008 competition, only JAMBOX[6] and $\mathsf{T_TT_2}$ could prove the system to be terminating (in 2009 it was not selected).

Example 7.1. The TRS Secret_06_TRS/gen-25 ($\mathcal{R}$ in the following) consists of three rewrite rules:

$$\begin{aligned} \mathsf{c}(\mathsf{c}(z,x,\mathsf{a}),\mathsf{a},y) &\to \mathsf{f}(\mathsf{f}(\mathsf{c}(y,\mathsf{a},\mathsf{f}(\mathsf{c}(z,y,x))))) \\ \mathsf{f}(\mathsf{f}(\mathsf{c}(\mathsf{a},y,z))) &\to \mathsf{b}(y,\mathsf{b}(z,z)) \\ \mathsf{b}(\mathsf{a},\mathsf{f}(\mathsf{b}(\mathsf{b}(z,y),\mathsf{a}))) &\to z \end{aligned}$$

The dependency graph contains one SCC, consisting of the following dependency pairs:

$$\begin{aligned} 1\colon \mathsf{C}(\mathsf{c}(z,x,\mathsf{a}),\mathsf{a},y) &\to \mathsf{C}(y,\mathsf{a},\mathsf{f}(\mathsf{c}(z,y,x))) \\ 2\colon \mathsf{C}(\mathsf{c}(z,x,\mathsf{a}),\mathsf{a},y) &\to \mathsf{C}(z,y,x) \end{aligned}$$

Hence termination of $\mathcal{R}$ is reduced to finiteness of the DP problem $(\mathcal{P},\mathcal{R},\mathcal{G})$ where $\mathcal{P}=\{1,2\}$ and $\mathcal{G}=(\mathcal{P},\mathcal{P}\times\mathcal{P})$. This problem is top-DP-bounded for rule 1; the compatible tree automaton computed by $\mathsf{T_TT_2}$ consists of 630 transitions and 47 states. Hence the DP processor of Theorem 5.12 is applicable. This results in the new DP problem $(\{2\},\mathcal{R},\mathcal{G}\setminus\{1\})$, which is proved finite by the subterm criterion with the simple projection $\pi(C)=1$.

7.2 Dependency Graphs

There are various ways to implement the (improved) dependency graph processors based on tree automata completion, ranging from checking single arcs to computing SCCs in between in order to reduce the number of arcs that have

[5] `http://termcomp.uibk.ac.at/`
[6] `http://joerg.endrullis.de/`

to be checked. In case of full termination, the following procedure turned out to be the most efficient. For every term $t \in \mathsf{rhs}(\mathcal{P})$, $\mathsf{T_TT_2}$ constructs a tree automaton $\mathcal{A}_t$ that is compatible with $\mathcal{R}$ and $L(t)$. Here $L(t) = \Sigma(\mathsf{ren}(t))$ in case of $\mathsf{DG_c}(\mathcal{P}, \mathcal{R})$ and $L(t) = \Sigma_{\#}(\mathsf{RFC}_{\{t\}}(\mathcal{P} \cup \mathcal{R})) \cap \Sigma(\mathsf{ren}(t))$ if $\mathsf{IDG_c}(\mathcal{P}, \mathcal{R})$ should be computed and $\mathcal{P} \cup \mathcal{R}$ is right-linear. During that process it is checked if there is a term $u \in \mathsf{lhs}(\mathcal{P})$ and a state substitution σ such that $u\sigma \in \mathcal{L}(\mathcal{A}_t)$. As soon as this condition evaluates to true, $\mathsf{T_TT_2}$ adds an arc from $s \to t$ to $u \to v$ for all terms s and v such that $s \to t$ and $u \to v$ are rules in $\mathcal{P}$. This procedure is repeated until for all $t \in \mathsf{rhs}(\mathcal{P})$, either $\mathcal{A}_t$ is compatible with $\mathcal{R}$ and $L(t)$ or an arc was added from $s \to t$ to $u \to v$ for all terms s and rules $u \to v \in \mathcal{P}$ such that $\mathsf{root}(t) = \mathsf{root}(u)$ and $s \to t$ belongs to $\mathcal{P}$. In case of innermost termination we use a simpler procedure. First of all $\mathsf{T_TT_2}$ tries to construct for each $t \in \mathsf{rhs}(\mathcal{P})$ a tree automaton $\mathcal{A}_t$ that is compatible with $\mathcal{R}$ and $L(t)$. Here $L(t) = \Sigma_{\#}(\mathsf{RFC}_{\{t\}}(\mathcal{P} \cup \mathcal{R})) \cap \Sigma_{\mathsf{NF}}(\mathsf{ren}(t), \mathcal{R})$ if $\mathcal{P} \cup \mathcal{R}$ is right-linear and $\mathsf{IDG^i_c}(\mathcal{P}, \mathcal{R})$ should be computed and $L(t) = \Sigma_{\mathsf{NF}}(\mathsf{ren}(t), \mathcal{R})$ in case of $\mathsf{DG^i_c}(\mathcal{P}, \mathcal{R})$. Afterwards $\mathsf{T_TT_2}$ computes the intersection of $\mathcal{L}(\mathcal{A}_t)$ and $\mathsf{NF}(\mathsf{lhs\text{-}linear}(\mathcal{R}))$ and checks for all terms $u \in \mathsf{lhs}(\mathcal{P})$ whether there is a state substitution σ such that $u\sigma \in \mathcal{L}(\mathcal{A}) \cap \mathsf{NF}(\mathsf{lhs\text{-}linear}(\mathcal{R}))$ or not. If $u\sigma \in \mathcal{L}(\mathcal{A}) \cap \mathsf{NF}(\mathsf{lhs\text{-}linear}(\mathcal{R}))$ an arc from $s \to t$ to $u \to v$, for all terms s and v such that $s \to t$ and $u \to v$ are rewrite rules in $\mathcal{P}$, is added. Otherwise the graph remains unchanged.

Another important point is the strategy used to solve compatibility violations. In $\mathsf{T_TT_2}$ we establish paths as described at the beginning of this chapter. A disadvantage of this strategy is that it can happen that the completion procedure does not terminate because new states are kept being added. Hence we have to set a time limit on the involved processors to avoid that the termination proving process does not proceed beyond the calculation of (improved) dependency graphs. Alternatively one could follow the approach based on approximation equations or approximation rules described in Section 3.3. However, our experiments showed that the former approach produces better over-approximations.

7.2.1 Full Termination

Below we report on the experiments we performed with $\mathsf{T_TT_2}$ on the 1370 TRSs in the full termination category of the TPDB that satisfy the variable condition. For the results in Tables 7.5 and 7.6 we used the following DP processors:

- t is a simple and fast approximation of the dependency graph processor of Definition 5.31 using root comparisons to estimate the dependency graph: an arc is added from a dependency pair α to a dependency pair β if and only if the root symbols of $\mathsf{rhs}(\alpha)$ and $\mathsf{lhs}(\beta)$ coincide.

- e represents the dependency graph processor with the estimation $\mathsf{DG_e}(\mathcal{P}, \mathcal{R})$ described in Definition 5.49. This is the default dependency graph processor in $\mathsf{T_TT_2}$ and AProVE.

- c corresponds to the dependency graph processor with the approximation $\mathsf{DG_c}(\mathcal{P}, \mathcal{R})$ of Definition 5.36.

Table 7.5: Summary dependency graph approximations

	t	e	c	r	$*$	
arcs removed	55	68	68	73	74	
# SCCs	4790	4904	4522	3970	4606	
# rules	26291	24046	21516	20026	23469	without
# successes	24	58	65	203	208	poly
average time	261	288	1078	1219	499	
# timeouts	0	0	32	76	0	
# successes	318	341	349	401	410	with
average time	293	289	479	456	362	poly
# timeouts	5	5	39	83	5	

Table 7.6: Summary dependency graph approximations with ur

	t	e	c	r	$*$	
arcs removed	55	67	68	74	74	
# SCCs	4790	4907	4486	3879	4550	
# rules	26291	24052	21269	19681	23360	without
# successes	24	56	65	218	220	poly
average time	271	296	1034	1165	504	
# timeouts	0	0	29	71	0	
# successes	318	338	349	408	414	with
average time	306	300	486	463	370	poly
# timeouts	5	5	35	77	5	

r corresponds to the improved dependency graph processor of Theorem 5.44. If $\mathcal{P}$ and $\mathcal{R}$ are right-linear, $\mathsf{IDG}(\mathcal{P},\mathcal{R})$ is estimated by $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$ of Definition 5.46 and in case that $\mathcal{P} \cup \mathcal{R}$ is non-right-linear, $\mathsf{DG_c}(\mathcal{P},\mathcal{R})$ is computed.

After applying the above processors we use the SCC processor of Definition 5.32. In the bottom third of the tables this is additionally followed by the reduction pair processor instantiated by linear polynomial interpretations with 0/1 coefficients (poly for short) [30].

In the top third of Tables 7.5 and 7.6 we list the average number of removed arcs (as percentage of the complete graph), the number of SCCs, and the number of rewrite rules in the computed SCCs. In the middle third we list the number of successful termination attempts, the average wall-clock time needed to compute the graphs (measured in milliseconds), and the number of timeouts. In the bottom third, where polynomial interpretations are in effect, the average time now refers to the time to prove termination.

The power of the new processors is apparent, although the difference with e decreases when other DP processors are in place. An obvious disadvantage of the new processors is the large number of timeouts. As explained earlier, this is mostly due to the unbounded number of new states to resolve compatibility violations during tree automata completion. Modern termination tools use a variety of techniques to prove finiteness of DP problems. So it is in general more important that the graph approximations used in the (improved) dependency graph processor terminate quickly rather than that they are powerful. Since the processors c and r seem to be quite fast when they terminate, an obvious idea is to equip each computation of a compatible tree automaton with a small time limit. Another natural idea is to limit the number of allowed compatibility violations. For instance, by reducing this number to 5 we can still prove termination of 150 TRSs with processor r while the number of timeouts is reduced from 76 to 9. Another strategy is to combine different graph approximations. This is shown in the columns of Tables 7.5 and 7.6 labeled $*$, which denotes the composition of t, e, c, and r with a time limit of 500 milliseconds for each of the latter three.

It is interesting to observe that r (and by extension $*$) is the only processor that benefits from usable rules. In case of e we anticipated such a behavior because similar as $\mathcal{U}(\mathcal{P},\mathcal{R})$, $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$ uses the function tcap to check via unification if there is an arc from a dependency pair α to a dependency pair β. So whenever there is a rewrite rule in $\mathcal{R}$ which causes $\mathsf{tcap}(\mathcal{R},\mathsf{rhs}(\alpha))$ and $\mathsf{lhs}(\beta)$ to be unifiable we know that this rule is usable. Of course this does not mean that $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$ is identical to $\mathsf{DG_e}(\mathcal{P},\mathcal{U}(\mathcal{P},\mathcal{R}))$. There are systems like the one of Example 7.2 where usable rules have a positive effect. The reason why the c-dependency graph does not benefit from usable rules is quite similar to the one we mentioned in connection with the estimated dependency graph. In most cases, only those rewrite rules cause compatibility violations which are also usable. Certainly, this is not always the case as shown in Subsection 5.4.2.

Example 7.2. Let $(\mathcal{P},\mathcal{R},\mathcal{G})$ be the DP problem with $\mathcal{P}$ consisting of the single rewrite rule $\mathsf{F}(\mathsf{a},x) \to \mathsf{F}(\mathsf{b},x)$, $\mathcal{R}$ consisting of the two rewrite rules $\mathsf{b} \to \mathsf{c}$ and $\mathsf{d} \to \mathsf{a}$, and $\mathcal{G} = (\mathcal{P},\mathcal{P} \times \mathcal{P})$. Obviously, $\mathsf{tcap}(\mathcal{R},\mathsf{F}(\mathsf{b},x)) = \mathsf{F}(y,z)$ and $\mathsf{tcap}(\mathcal{R}^{-1},\mathsf{F}(\mathsf{a},x)) = \mathsf{F}(y,z)$. Since $\mathsf{F}(y,z)$ unifies with $\mathsf{F}(\mathsf{a},x)$ and $\mathsf{F}(\mathsf{b},x)$ unifies with $\mathsf{F}(y,z)$ we know that $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$ contains an arc from $\mathsf{F}(\mathsf{a},x) \to \mathsf{F}(\mathsf{b},x)$ to itself. In contrast, $\mathsf{DG_e}(\mathcal{P},\mathcal{U}(\mathcal{P},\mathcal{R}))$ does not contain such an arc. We have $\mathcal{U}(\mathcal{P},\mathcal{R}) = \{\mathsf{b} \to \mathsf{c}\}$ and hence $\mathsf{tcap}(\mathcal{R}^{-1},\mathsf{F}(\mathsf{a},x)) = \mathsf{F}(\mathsf{a},y)$. Since $\mathsf{F}(\mathsf{b},x)$ does not unify with $\mathsf{F}(\mathsf{a},y)$, $\mathsf{DG_e}(\mathcal{P},\mathcal{U}(\mathcal{P},\mathcal{R}))$ is empty.

We also implemented the approximations based on tree automata and regularity preservation described in Subsection 5.3.4. The results are summarized in Tables 7.7 and 7.8. Thereby s represents the dependency graph processor estimated by $\mathsf{DG_s}(\mathcal{P},\mathcal{R})$, nv defines the dependency graph processor using $\mathsf{DG_{nv}}(\mathcal{P},\mathcal{R})$ to approximate $\mathsf{DG}(\mathcal{P},\mathcal{R})$, and g corresponds to the dependency graph processor estimated by $\mathsf{DG_g}(\mathcal{P},\mathcal{R})$. It is apparent that these approximations are too time-consuming to be of any use in automatic termination provers. This is implicitly indicated by the high average time that is needed to compute the graphs and the huge number of timeouts. As an immediate

Table 7.7: Summary ϕ-approximated dependency graphs

	c	s	nv	g	
arcs removed	68	64	60	48	
# SCCs	4522	2407	1600	63	
# rules	21516	6345	3898	105	without
# successes	65	58	64	35	poly
average time	1078	4515	5119	5051	
# timeouts	32	232	468	1275	
# successes	349	315	295	56	with
average time	479	1923	2317	4962	poly
# timeouts	39	233	468	1275	

Table 7.8: Summary ϕ-approximated dependency graphs with ur

	c	s	nv	g	
arcs removed	68	64	61	49	
# SCCs	4486	2438	1777	259	
# rules	21269	6425	4207	432	without
# successes	65	56	61	54	poly
average time	1034	3869	4180	2320	
# timeouts	29	222	421	1112	
# successes	349	312	297	131	with
average time	486	1847	1957	1801	poly
# timeouts	35	222	420	1112	

consequence it would not make much sense to equip those approximations with a small time limit of 500 milliseconds as we did with the DP processors c and r in the columns labeled with $*$.

One advantage of more powerful (improved) dependency graph approximations is that termination proofs can get much simpler.

Example 7.3. The TRS Endrullis_06/quadruple1 ($\mathcal{R}$ in the following) consists of the following rewrite rule:

$$\mathsf{p}(\mathsf{p}(\mathsf{b}(\mathsf{a}(x)), y), \mathsf{p}(z, u)) \to \mathsf{p}(\mathsf{p}(\mathsf{b}(z), \mathsf{a}(\mathsf{a}(\mathsf{b}(y)))), \mathsf{p}(u, x))$$

To prove termination of $\mathcal{R}$ using dependency pairs, we transform $\mathcal{R}$ into the initial DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ where $\mathcal{P} = \mathsf{DP}(\mathcal{R})$ consists of the rewrite rules

$$1\colon \mathsf{P}(\mathsf{p}(\mathsf{b}(\mathsf{a}(x)), y), \mathsf{p}(z, u)) \to \mathsf{P}(\mathsf{p}(\mathsf{b}(z), \mathsf{a}(\mathsf{a}(\mathsf{b}(y)))), \mathsf{p}(u, x))$$
$$2\colon \mathsf{P}(\mathsf{p}(\mathsf{b}(\mathsf{a}(x)), y), \mathsf{p}(z, u)) \to \mathsf{P}(\mathsf{b}(z), \mathsf{a}(\mathsf{a}(\mathsf{b}(y))))$$
$$3\colon \mathsf{P}(\mathsf{p}(\mathsf{b}(\mathsf{a}(x)), y), \mathsf{p}(z, u)) \to \mathsf{P}(u, x)$$

and $\mathcal{G} = \mathcal{P} \times \mathcal{P}$. Applying the improved dependency graph processor produces the new DP problem $(\mathcal{P}, \mathcal{R}, \mathsf{IDG_c}(\mathcal{P}, \mathcal{R}))$ where $\mathsf{IDG_c}(\mathcal{P}, \mathcal{R}) = \varnothing$. It follows that the initial DP problem is finite and hence that $\mathcal{R}$ is terminating. If we use $\mathsf{DG_e}(\mathcal{P}, \mathcal{R})$ instead of $\mathsf{IDG_c}(\mathcal{P}, \mathcal{R})$ we obtain the DP problem $(\mathcal{P}, \mathcal{R}, \mathsf{DG_e}(\mathcal{P}, \mathcal{R}))$ where $\mathsf{DG_e}(\mathcal{P}, \mathcal{R})$ looks as follows:

$$\begin{tikzcd} 2 & 1 \arrow[l] \arrow[r] \arrow[loop above] & 3 \end{tikzcd}$$

By applying the SCC processor of Definition 5.32 we end up with the non-empty DP problem $(\{1\}, \mathcal{R}, \mathsf{DG_e}(\mathcal{P}, \mathcal{R}) \setminus \{2, 3\})$. So termination of $\mathcal{R}$ cannot be shown that easily. This is reflected in the 2009 edition of the international termination competition: AProVE combined a variety of processors to infer termination within 2.1 seconds, JAMBOX proved termination of $\mathcal{R}$ within 2.38 seconds by using linear matrix interpretations up to dimension 2, and TTT2 used match-bounds together with right-hand sides of forward closures to prove termination within 242 milliseconds. All other tools could not show the termination of the TRS $\mathcal{R}$.

7.2.2 Innermost Termination

In this subsection we report on the experiments we performed with TTT2 on the 358 TRSs in the innermost termination category of the TPDB. For the results in Tables 7.9 and 7.10 we used the innermost counterparts of the DP processors t, e, c, and r of the previous subsection:

- t approximates the innermost dependency graph processor of Definition 5.55 using root comparisons.
- e estimates the innermost dependency graph processor with $\mathsf{DG^i_e}(\mathcal{P}, \mathcal{R})$ described in Definition 5.63. This is the default innermost dependency graph processor in TTT2 and AProVE.
- c approximates the innermost dependency graph processor with $\mathsf{DG^i_c}(\mathcal{P}, \mathcal{R})$ of Definition 5.61.
- r represents the improved innermost dependency graph processor of Theorem 5.59 where $\mathsf{IDG^i_c}(\mathcal{P}, \mathcal{R})$ of Definition 5.61 is computed if $\mathcal{P}$ and $\mathcal{R}$ are right-linear and $\mathsf{DG^i_c}(\mathcal{P}, \mathcal{R})$ otherwise.

Similar as for full termination we use $*$ to denote the composition of the innermost DP processors t, e, c, and r where each of the latter three is equipped with a time limit of 500 milliseconds. After applying the above processors we use the SCC processor of Definition 5.32. In the bottom third of the tables this is additionally followed by the application of poly.

The power of the new processors is clearly visible, although the difference with e is not that large as in the case of full termination. A serious problem of the innermost processors c and r is that the normal form computations require

Table 7.9: Summary innermost dependency graph approximations

	t	e	c	r	∗	
arcs removed	69	78	72	79	81	
# SCCs	2751	2485	198	102	2674	
# rules	16419	13927	543	305	15686	without
# successes	9	24	32	53	51	poly
average time	421	1593	4726	4146	1375	
# timeouts	0	2	239	248	0	
# successes	74	92	66	65	99	with
average time	516	493	4159	2609	962	poly
# timeouts	5	9	239	248	5	

Table 7.10: Summary innermost dependency graph approximations with ur

	t	e	c	r	∗	
arcs removed	69	78	70	77	81	
# SCCs	2751	2483	206	108	2674	
# rules	16419	13607	547	305	15692	without
# successes	9	23	28	50	51	poly
average time	547	1470	3724	2632	1459	
# timeouts	0	3	237	246	0	
# successes	74	91	63	63	99	with
average time	522	498	3779	1759	944	poly
# timeouts	5	9	237	246	5	

additional resources which lead to a huge number of timeouts. As a side effect the number of innermost termination proofs of the c and r processor are much lower than the number of successful innermost termination proofs of e, if polynomial interpretations are in effect. To control the number of timeouts one can use the same techniques as discussed in Subsection 7.2.1.

Another interesting observation is that none of the processors can really benefit from usable rules. Of course there are TRSs like AG01_innermost/#4.25 where $\mathsf{DG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$ and $\mathsf{IDG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$ contain less arcs if usable rules are computed beforehand. Nevertheless, it seems that the negative effect of usable rules, indicated in Subsection 5.4.2, is more serious than assumed.

7.3 Complexity Analysis

In the following section we report on the experiments we performed with CaT on the 1172 TRSs in the complexity category of the TPDB that are non-duplicating.[7] Our results are summarized in Tables 7.11 and 7.12. In Table 7.11 we are concerned with derivational complexities whereas in Table 7.12 we investigate the runtime complexities of the considered TRSs. We list the number of successful termination attempts (inducing a polynomial complexity), the number of TRSs that admit a linear, quadratic, or cubic complexity, and the average system time needed to prove termination (measured in milliseconds). To compute upper complexity bounds we used the following techniques:

b represents the ordinary match-bound technique based on Theorems 4.4 and 4.15. To estimate the derivational complexity of a TRS we have chosen as initial language the set of all ground terms and to evaluate the runtime complexity of a TRS we considered the set of all constructor-based terms.

p corresponds to the CP processor based on strongly linear interpretations with 0/1 coefficients [61].

m abbreviates two CP processors based on triangular matrix interpretations of dimension two and three with 0/1 coefficients [61].

r represents the CP processor based on Theorems 6.23 and 6.24. Similar as for match(-raise)-bounds we considered as initial language the set of all ground terms to estimate the derivational complexity of a TRS and the set of all constructor-based terms to evaluate the runtime complexity of a TRS.

Note that we do not mention the number of timeouts because all techniques have been equipped with a small time limit. This is necessary because linear complexity proofs are preferred to quadratic complexity proofs etc. As a consequence the used strategies always terminate. The use of compatible tree automata is indicated by c and we write qc to denote that quasi-compatible tree automata have been constructed. To handle raise-rules we used the implicit approach.

Our experiments confirm the conjecture that match(-raise)-RT-bounds are more powerful than match(-raise)-bounds. Although there is only a minor difference between the two approaches, there are at least some TRSs like Various_04/24 or Zantema_04/z024 that can be proved polynomially terminating if we use match-RT-bounds instead of match-bounds. Quite interesting is that some of these systems do not admit a linear complexity. So match(-raise)-RT-bounds are not only useful to prove linear complexity bounds. The reason for this behavior can be traced back to the complexity framework which makes it possible to combine different techniques to prove complexity bounds. Another interesting observation is the tremendous increase of linear complexity proofs if

[7]If we want to compute the derivational complexity of a TRS, non-duplication is no real restriction because any duplicating TRS admits exponential derivational complexity. In case of runtime complexity the situation is different. There are TRSs which are duplicating but admit a polynomial runtime complexity. Nevertheless, we had to restrict our experiments to non-duplicating TRSs because match(-raise)-bounds and in the sequel match(-raise)-RT-bounds cannot be used to prove polynomial upper complexity bounds of duplicating TRSs.

Table 7.11: Summary derivational complexity

		using c		using qc	
	pm	bpm	bprm	bpm	bprm
# successes	170	270	276	268	274
# linear	38	171	179	171	180
# quadratic	110	85	84	83	82
# cubic	22	14	13	14	12
average time	514	2344	4174	2339	4120

Table 7.12: Summary runtime complexity

		using c		using qc	
	pm	bpm	bprm	bpm	bprm
# successes	170	921	928	919	925
# linear	38	907	916	905	913
# quadratic	110	13	11	13	11
# cubic	22	1	1	1	1
average time	514	276	390	276	385

we switch from derivational complexity to runtime complexity. The reasons for this gain in power are that the complexity category of the TPDB contains many TRSs where all function symbols are defined (such TRSs admit most times a linear or even constant complexity, especially if every left-hand side consists of at least two defined symbols) and that the match-bound technique allows us to investigate the termination of a given TRS with respect to an arbitrary regular language. So it is not surprising that match(-raise)-bounds and match(-raise)-RT-bounds are so far the most powerful techniques to prove linear runtime complexity.

In the 2009 edition of the international termination competition none of the tools could prove an upper complexity bound for the TRS Transformed_CSR_04/Ex16_Luc06_GM. Using match-raise-RT-bounds CaT is now able to show that this TRS admits a quadratic derivational complexity.

Example 7.4. We compute the derivational complexity of the rewrite system Transformed_CSR_04/Ex16_Luc06_GM ($\mathcal{R}$ in the following) over the signature $\mathcal{F} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{f}, \mathsf{g}, \mathsf{mark}\}$ which consists of the following rewrite rules:

$$\begin{array}{lll} \mathsf{c} \to \mathsf{a} & \mathsf{g}(x, y) \to \mathsf{f}(x, y) & \mathsf{mark}(\mathsf{a}) \to \mathsf{a} \\ \mathsf{c} \to \mathsf{b} & \mathsf{g}(x, x) \to \mathsf{g}(\mathsf{a}, \mathsf{b}) & \mathsf{mark}(\mathsf{b}) \to \mathsf{c} \\ & & \mathsf{mark}(\mathsf{f}(x, y)) \to \mathsf{g}(\mathsf{mark}(x), y) \end{array}$$

By using the CP processor p, we can transform the initial CP problem consisting of the relative TRS $\mathcal{R}/\varnothing$ into the CP problem $(\mathcal{R}'/\mathcal{S}', \mathcal{T}(\mathcal{F}), f')$ where the TRS

$\mathcal{R}'$ consist of the rewrite rules

$$\mathsf{g}(x,x) \to \mathsf{g}(\mathsf{a},\mathsf{b}) \qquad\qquad \mathsf{mark}(\mathsf{f}(x,y)) \to \mathsf{g}(\mathsf{mark}(x),y)$$

and $\mathcal{S}' = \mathcal{R} \setminus \mathcal{R}'$. Since p induces linear complexity bounds we have $f'(n) = n$ for all $n \in \mathbb{N}$. After that we can apply the CP processor r to show that the derivational complexity induced by the rewrite rule $\mathsf{g}(x,x) \to \mathsf{g}(\mathsf{a},\mathsf{b})$ is at most linear. So we end up with the CP problem $(\mathcal{R}''/\mathcal{S}'', \mathcal{T}(\mathcal{F}), f'')$ where the TRS $\mathcal{R}''$ consist of the rewrite rule $\mathsf{mark}(\mathsf{f}(x,y)) \to \mathsf{g}(\mathsf{mark}(x),y)$, $\mathcal{S}'' = \mathcal{R} \setminus \mathcal{R}''$, and $f''(n) = f'(n) + n = 2n$ for all $n \in \mathbb{N}$. Finally, by applying the CP processor m with triangular matrix interpretations of dimension two we obtain the CP problem $(\varnothing/\mathcal{R}, \mathcal{T}(\mathcal{F}), f)$ where $f(n) = f''(n) + n^2 = n^2 + 2n$ for all $n \in \mathbb{N}$. It follows that the derivational complexity of $\mathcal{R}$ can be bounded by a quadratic polynomial. Note that the quadratic bound is tight as $\mathcal{R}$ admits derivations

$$\mathsf{mark}^n(x)\sigma^m \to^m \mathsf{mark}^{n-1}(x)\tau^m\theta \to^m \mathsf{mark}^{n-1}(x)\sigma^m\theta \to^{2m(n-1)} x\sigma^m\theta^n$$

of length $2mn$ where $\sigma = \{x \mapsto \mathsf{f}(x,y)\}$, $\tau = \{x \mapsto \mathsf{g}(x,y)\}$, and $\theta = \{x \mapsto \mathsf{mark}(x)\}$. Last but not least we remark that none of the involved techniques can establish an upper bound on its own. In case of the match-bound technique this follows from the fact that $\mathcal{R}$ admits quadratic derivational complexity. The same reason also holds for p because strongly linear interpretations induce linear complexity bounds. Finally, m fails because it cannot orient the rule $\mathsf{g}(x,x) \to \mathsf{g}(\mathsf{a},\mathsf{b})$.

7.4 Summary

In this chapter we reported on the extensive experiments we conducted. We showed that termination tools can benefit from the termination techniques presented in the previous chapters.

The main reason why the presented techniques are quite successful is that they can be adapted to any situation by configurating the initial language. For instance, in case of e(-raise)(-DP)-bounds we used right-hand sides of forward closures to prove the termination of rewrite systems. Likewise, we used forward closures to go beyond dependency graphs. For complexity analysis we could use the mentioned ability to configurate match(-raise)(-RT)-bounds in such a way that they are especially suited to prove runtime complexity bounds (we have chosen the set of all constructor-based terms as initial language). So it is not astonishing that we sometimes got a large increase of the number of termination or complexity proofs.

A serious disadvantage of all methods based on tree automata completion is that they either succeed or never terminate. So to ensure a successful integration into a termination or complexity prover it is necessary to equip each technique with a time limit. In $\mathsf{T_TT_2}$ and CaT we usually choose a small time limit of at most 5 seconds since the employed methods seem to be quite fast if they succeed.

Chapter 8

Conclusion

In this thesis we have shown that tree automata techniques, especially *tree automata completion*, can efficiently be used to check if a given TRS is terminating. As starting point we considered the *match-bound technique* which uses tree automata techniques, in particular, tree automata completion to obtain termination certificates. Since tree automata completion is mainly justified for left-linear rewrite systems, the initially proposed match-bound technique could only be used for this restricted class of TRSs. To eliminate this burden we presented in Chapter 3 a new and elegant approach to cope with non-left-linear TRSs. The key to this extension is the use of *quasi-deterministic* tree automata instead of non-deterministic tree automata during the completion process. However, to extend the match-bound technique to non-left-linear rewrite systems it does not suffice to broaden tree automata completion. The reason is that the theory on which the method is based is incorrect for non-left-linear TRSs. To overcome this problem we introduced so called *raise-rules* in Chapter 4. Finally we showed how to strengthen the method by taking *forward closures* into account.

Chapter 5 was devoted to the so called *dependency pair framework*. First of all we showed how the match-bound technique can be successfully integrated into this framework. For that purpose we introduced two new enrichments which take care of the special properties of dependency pair problems. Afterwards we illustrated how tree automata completion can be used to approximate *dependency graphs*. Thereby we used forward closures to remove arcs of the exact dependency graph. Last but not least we combined all developed DP processors with *usable rules*.

After successfully performing termination analysis it is natural to determine the *complexity* of the given TRSs. To show feasible upper complexity bounds only a few techniques such as match-bounds and triangular matrix interpretations are known. Typically, those techniques are used in a direct way only. So a single termination technique has to orient all rules in one go. In Chapter 6 we presented a modular approach, the so called *complexity framework*, which computes the complexity of a given TRS using relative rewriting. We also showed how the match-bound technique can be integrated into this setting.

Finally in Chapter 7 we reported on the extensive experiments we conducted. We showed that termination tools can benefit from the termination techniques presented in the previous chapters. We also figured out that the main reason why the presented methods are quite successful is that they can be adapted to any situation by configurating the initial language in an appropriate way. A serious disadvantage of the presented techniques is that they either succeed or never terminate. So to successfully use them in a termination or complexity prover it is necessary to equip each method with a time limit.

Bibliography

[1] Tomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1–2):133–178, 2000.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[3] Françoise Bellegarde and Pierre Lescanne. Termination by completion. *Applicable Algebra in Engineering, Communication and Computing*, 1(2):79–96, 1990.

[4] Yohan Boichut, Thomas Genet, Thomas Jensen, and Luka Le Roux. Rewriting approximations for fast prototyping of static analyzers. In *Proceedings of the 18th International Conference on Rewriting Techniques and Applications (RTA 2007)*, volume 4533 of *Lecture Notes in Computer Science*, pages 48–62, 2007.

[5] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[6] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Available from `tata.gforge.inria.fr`, 2007.

[7] Nachum Dershowitz. Termination of linear rewriting systems (preliminary version). In *Proceedings of the 8th International Colloquium on Automata, Languages and Programming (ICALP 1981)*, volume 115 of *Lecture Notes in Computer Science*, pages 448–458, 1981.

[8] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[9] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[10] Iréne Durand and Aart Middeldorp. Decidable call-by-need computations in term rewriting. *Information and Computation*, 196(2):95–126, 2005.

[11] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV 2006)*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94, 2006.

[12] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518, 2004.

[13] Jörg Endrullis, Dieter Hofbauer, and Johannes Waldmann. Decomposing terminating rewrite relations. In *Proceedings of the 8th International Workshop on Termination (WST 2006)*, pages 39–43, 2006.

[14] Guillaume Feuillade, Thomas Genet, and Valérie Viet Triem Tong. Reachability analysis over term rewriting systems. *Journal of Automated Reasoning*, 33(3–4):341–383, 2004.

[15] Jean Gallier and Ronald Book. Reductions in tree replacement systems. *Theoretical Computer Science*, 37:123–150, 1985.

[16] Adrià Gascón, Guillem Godoy, and Florent Jacquemard. Closure of tree automata languages under innermost rewriting. *Electronic Notes in Theoretical Computer Science*, 237:23–38, 2009. Proceedings of the 8th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2008).

[17] Thomas Genet. Decidable approximations of sets of descendants and sets of normal forms (extended version). Technical Report RR-3325, Institut National de Recherche en Informatique et Automatique (INRIA), 1997.

[18] Thomas Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Proceedings of the 9th International Conference on Rewriting Techniques and Applications (RTA 1998)*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165, 1998.

[19] Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification. In *Proceedings of the 17th International Conference on Automated Deduction (CADE 2000)*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 271–290, 2000.

[20] Thomas Genet and Valérie Viet Triem Tong. Reachability analysis of term rewriting systems with Timbuk. In *Proceedings of the 8th International Conference on Logic Programming and Automated Reasoning (LPAR 2001)*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 695–706, 2001.

[21] Alfons Geser, Dieter Hofbauer, and Johannes Waldmann. Match-bounded string rewriting systems. *Applicable Algebra in Engineering, Communication and Computing*, 15(3–4):149–171, 2004.

[22] Alfons Geser, Dieter Hofbauer, and Johannes Waldmann. Termination proofs for string rewriting systems via inverse match-bounds. *Journal of Automated Reasoning*, 34(4):365–385, 2005.

[23] Alfons Geser, Dieter Hofbauer, Johannes Waldmann, and Hans Zantema. Finding finite automata that certify termination of string rewriting systems. *International Journal of Foundations of Computer Science*, 16(3):471–486, 2005.

[24] Alfons Geser, Dieter Hofbauer, Johannes Waldmann, and Hans Zantema. On tree automata that certify termination of left-linear term rewriting systems. *Information and Computation*, 205(4):512–534, 2007.

[25] Oliver Geupel. Overlap closures and termination of term rewriting systems. Technical Report MIP-8922, Universität Passau, 1989.

[26] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 281–286, 2006.

[27] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proceedings of the 11th International Conference on Logic Programming and Automated Reasoning (LPAR 2004)*, volume 3425 of *Lecture Notes in Artificial Intelligence*, pages 301–331, 2004.

[28] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proceedings of the 5th International Workshop on Frontiers of Combining Systems (FroCoS 2005)*, volume 3717 of *Lecture Notes in Artificial Intelligence*, pages 216–231, 2005.

[29] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Improving dependency pairs. In *Proceedings of the 10th International Conference on Logic Programming and Automated Reasoning (LPAR 2003)*, volume 2850 of *Lecture Notes in Artificial Intelligence*, pages 167–182, 2003.

[30] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

[31] Rémi Gilleron and Sophie Tison. Regular tree languages and rewrite systems. *Fundamenta Informaticae*, 24(1–2):157–175, 1995.

[32] Nao Hirokawa and Aart Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1–2):172–199, 2005.

[33] Nao Hirokawa and Aart Middeldorp. Tyrolean termination tool: Techniques and features. *Information and Computation*, 205(4):474–511, 2007.

[34] Nao Hirokawa and Georg Moser. Automated complexity analysis based on the dependency pair method. In *Proceedings of the 4th International*

Joint Conference on Automated Reasoning (IJCAR 2008), volume 5195 of *Lecture Notes in Computer Science*, pages 364–379, 2008.

[35] Nao Hirokawa and Georg Moser. Complexity, graphs, and the dependency pair method. In *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2008)*, volume 5330 of *Lecture Notes in Artificial Intelligence*, pages 652–666, 2008.

[36] Dieter Hofbauer and Clemens Lautemann. Termination proofs and the length of derivations (preliminary version). In *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications (RTA 1989)*, volume 355 of *Lecture Notes in Computer Science*, pages 167–177, 1989.

[37] Dieter Hofbauer and Johannes Waldmann. Deleting string rewriting systems preserve regularity. *Theoretical Computer Science*, 327(3):301–317, 2004.

[38] Donald Knuth and Peter Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[39] Adam Koprowski and Johannes Waldmann. Arctic termination ... below zero. In *Proceedings of the 19th International Conference on Rewriting Techniques and Applications (RTA 2008)*, volume 5117 of *Lecture Notes in Computer Science*, pages 202–216, 2008.

[40] Martin Korp and Aart Middeldorp. Proving termination of rewrite systems using bounds. In *Proceedings of the 18th International Conference on Rewriting Techniques and Applications (RTA 2007)*, volume 4533 of *Lecture Notes in Computer Science*, pages 273–287, 2007.

[41] Martin Korp and Aart Middeldorp. Match-bounds with dependency pairs for proving termination of rewrite systems. In *Proceedings of the 2nd International Conference on Language and Automata Theory and Applications (LATA 2008)*, volume 5196 of *Lecture Notes in Computer Science*, pages 321–332, 2008.

[42] Martin Korp and Aart Middeldorp. Beyond dependency graphs. In *Proceedings of the 22nd International Conference on Automated Deduction (CADE 2009)*, volume 5663 of *Lecture Notes in Computer Science*, pages 339–354, 2009.

[43] Martin Korp and Aart Middeldorp. Match-bounds revisited. *Information and Computation*, 207(11):1259–1283, 2009.

[44] Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean termination tool 2. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA 2009)*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304, 2009.

[45] Keiichirou Kusakari. *Termination, AC-Termination and Dependency Pairs of Term Rewriting Systems.* PhD thesis, Japan Advanced Institute of Science and Technology (JAIST), 2000.

[46] Keiichirou Kusakari and Yoshihito Toyama. On proving AC-termination by AC-dependency pairs. Research Report IS-RR-98-0026F, School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), 1998.

[47] Zohar Manna and Stephen Ness. On the termination of Markov algorithms. In *Proceedings of the 3rd Hawaii International Conference on System Science (HICSS 1970)*, pages 789–792, 1970.

[48] Aart Middeldorp. Approximating dependency graphs using tree automata techniques. In *Proceedings of the 1st International Joint Conference on Automated Reasoning (IJCAR 2001)*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 593–610, 2001.

[49] Aart Middeldorp. Approximations for strategies and termination. *Electronic Notes in Theoretical Computer Science*, 70(6):1–20, 2002. Proceedings of the 2nd International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002).

[50] Georg Moser, Andreas Schnabl, and Johannes Waldmann. Complexity analysis of term rewriting based on matrix and context dependent interpretations. In *Proceedings of the 28th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, volume 1762 of *Dagstuhl Research Online Publication Server*, pages 304–315, 2008.

[51] Takashi Nagaya and Yoshihito Toyama. Decidability for left-linear growing term rewriting systems. *Information and Computation*, 178(2):499–514, 2002.

[52] Frédéric Oehl, Gérard Cece, Olga Kouchnarenko, and David Sinclair. Automatic approximation for the verification of cryptographic protocols. In *Proceedings of the 1st International Conference on Formal Aspects of Security (FASec 2002)*, volume 2629 of *Lecture Notes in Artificial Intelligence*, pages 33–48, 2003.

[53] David Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304.

[54] Terese. *Term Rewriting Systems.* Cambridge University Press, 2003.

[55] René Thiemann. *The DP Framework for Proving Termination of Term Rewriting.* PhD thesis, Rheinisch-Westfaelische Technische Hochschule Aachen (RWTH Aachen), 2007. Available as technical report AIB-2007-17.

[56] Axel Thue. *Probleme über Veränderungen von Zeichenreihen nach gegebenen Regeln.* Number 10 in Matematisk-Naturvidenskabelig Klasse. Skrifter utgit av Videnskapsselskapet i Kristiania, 1914.

[57] Alan Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[58] Alan Turing. Computability and λ-definability. *Journal of Symbolic Logic*, 2(4):153–163, 1937.

[59] Johannes Waldmann. Matchbox: A tool for match-bounded string rewriting. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA 2004)*, volume 3091 of *Lecture Notes in Computer Science*, pages 85–94, 2004.

[60] Johannes Waldmann. Weighted automata for proving termination of string rewriting. *Journal of Automata, Languages and Combinatorics*, 12(4):545–570, 2007.

[61] Harald Zankl and Martin Korp. Modular complexity analysis via relative complexity. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA 2010)*, 2010. To be published.

[62] Hans Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24(1–2):89–105, 1995.

[63] Hans Zantema. Termination of rewriting proved automatically. *Journal of Automated Reasoning*, 34(2):105–139, 2005.

Appendix A

Termination and Complexity Tools

In this chapter we summarize the main design issues, features, and successes of the termination prover TTT2 and the complexity tool CaT. In Section A.1 we describe the architecture and most important features of TTT2. In particular, we explain the strategy language of TTT2 which gives the user full control over the implemented termination methods. Additionally we also list the strategies that have been used to conduct the experiments presented in Sections 7.1 and 7.2. Afterwards, in Section A.2, we describe the most important features of CaT. Similar as for TTT2 we explain its strategy language, the output, as well as the strategies that have been used to obtain the experimental data shown in Section 7.3.

Most of the information presented in this appendix appeared already in the conference paper [44] which was related to version 1.0 of TTT2. In the following we updated the information to conform to version 1.05 of TTT2.

A.1 The Termination Tool TTT2

TTT2 is the completely redesigned successor of the *Tyrolean Termination Tool* (TTT for short) [33]. It is a tool for automatically proving termination of TRSs, based on the dependency pair framework [27, 30, 33, 55]. It incorporates several novel methods like increasing interpretations, match-bounds, uncurrying, and outermost loops. It produces readable output and has a simple web interface. Precompiled binaries, sources and documentation of TTT2 are available at the following web-site:

`http://cl-informatik.uibk.ac.at/software/ttt2/`

In contrast to its predecessor, TTT2 is open source, published under terms of the GNU Lesser General Public License. In the remainder of this section we refer to version 1.05 of the tool.

A.1.1 Design and Execution

The tool is written in OCaml[1] and consists of about 32,000 lines of code. Approximately 16% are dedicated to provide some general useful functions and

[1]`http://caml.inria.fr/`

data structures. Another 27% are used to implement the rewriting library which deals with terms and rules, the automata library which provides basic functions concerning tree automata and tree transducers, and the logic library which interfaces the used SAT and SMT solvers. The biggest fragment—about 47%—is used to implement the accessible termination methods and the strategy language. The rest (about 10%) is concerned with input and output. Since our tool provides several techniques that modify a termination problem by transforming it into different problem domains, $\mathsf{T_TT_2}$ interfaces the SAT solver MiniSat [12] and the SMT solver Yices [11]. For interfacing C code the third party contribution CamlIDL[2] is needed. The use of monads to implement the strategy language and several other parts of the tool, allows a clean and abstract treatment of the internal prover state in a purely functional way. Additionally, monads facilitate changes (like the integration of a new termination method).

Besides the actual termination prover, we provide the following libraries:

- `util` extends the functionality of several modules from the standard OCaml library. Modules for graph manipulation, advanced process and timer handling, as well as monads are also included.
- `parsec` is an OCaml port of the Haskell parsec[3] library, which provides an implementation of a functional combinator parser library.
- `rewriting` provides types and functions dealing with terms, substitutions, contexts, TRSs, etc. The functionality is not only aimed at termination. For instance the computation of overlaps and normal forms is also supported.
- `logic` provides an OCaml interface that abstracts over the two constraint solvers MiniSat and Yices. To this end arithmetical formulas are encoded in an intermediate datatype. When solving the constraints the user specifies the back-end. In the case of MiniSat, additional information (how many bits are used to represent numbers and intermediate results) can be provided. Afterwards the propositional formula is transformed into conjunctive normal form by a satisfiability-preserving transformation [53]. Yices, on the other hand, does neither require the number of bits as a parameter nor the transformation due to built-in support for linear arithmetic and formulas not in conjunctive normal form.
- `automata` provides an implementation of tree automata and tree transducers. The emphasis is put on functions which are especially useful in connection with tree automata completion.
- `processors` collects the numerous (non-)termination methods.
- `ttt2` contains the strategy language and connects the preceding libraries.

In order to run $\mathsf{T_TT_2}$ from the command line, the user can either download the source code from the $\mathsf{T_TT_2}$ web page and install it following the installation

[2]`http://caml.inria.fr/pub/old_caml_site/camlidl/`

[3]`http://legacy.cs.uu.nl/daan/parsec.html`

guidelines or alternatively download the binary of the latest version of TTT2. After a successful installation, TTT2 can be started via the command

```
./ttt2 [options] <file> [timeout]
```

where `[options]` denotes a list of command line options, `<file>` specifies the name of the file containing the TRS of which termination should be proved, and `[timeout]`—a floating point number—defines the time limit for proving termination of the given TRS. The TRS must adhere to the termination problem database format.[4] The timeout is optional. To get a complete list of the command line options of TTT2 execute the command `./ttt2 --help`.

A.1.2 The Strategy Language

As mentioned in the introduction, TTT2 is designed according to the dependency pair framework which ensures that all methods are implemented in a modular way. In order to combine these methods in a flexible manner, TTT2 provides a *strategy language*. In the following the most important constructs of this language are explained. For further information please consult the documentation of TTT2 by executing the command `./ttt2 --help`.

Syntax

The operators provided by the strategy language can be divided into three classes: *combinators*, *iterators*, and *specifiers*. Combinators are used to combine two strategies whereas iterators are used to repeat a given strategy a designated number of times. In contrast, specifiers are used to control the behavior of strategies. The most common combinators are the infixes '`;`', '`|`', and '`||`'. The most common iterators are the postfixes '`?`', '`+`', and '`*`'. The most common specifiers are '{ }' and '`[`f`]`' (written postfix), where f denotes some floating point number. In order to obtain a well-formed strategy s, these operators have to be combined according to the grammar

$$s ::= m \mid (s) \mid s\texttt{;}s \mid s\texttt{|}s \mid s\texttt{||}s \mid s\texttt{?} \mid s\texttt{+} \mid s\texttt{*} \mid s\texttt{[}f\texttt{]} \mid \{s\}o$$

where m denotes any available method and o any available modifier of TTT2 (possibly followed by some flags to adjust the considered technique). For the moment we just want to remark that the main difference between methods and modifiers is that the former operate on one termination problem whereas the latter get two termination problems as input (the current problem as well as some old problem). In order to avoid unnecessary parentheses, the following precedence is used: $\texttt{?}, \texttt{+}, \texttt{*}, \texttt{[}f\texttt{]} \; > \; \texttt{;} \; > \; \texttt{|}, \texttt{||}$.

Semantics

In the remainder of this subsection we use the notion *termination problem* to denote a TRS, a DP problem, or a relative termination problem. We call a termination problem *terminating* if the underlying TRS (DP problem, relative

[4] `http://www.termination-portal.org/`

termination problem) is terminating (finite, relative terminating). A strategy works on a termination problem. Whenever TTT2 executes a strategy, internally, a so called proof object is constructed which represents the actual termination proof. Depending on the shape of the resulting proof object after applying a strategy s, we say that s *succeeded* or s *failed*.

This should not be confused with the possible answers of the prover: `YES`, `NO`, and `MAYBE`. Here `YES` means that termination could be proved, `NO` indicates a successful non-termination proof, and `MAYBE` refers to the case when termination could neither be proved nor disproved. On success of a strategy s it depends on the internal proof object whether the final answer is `YES` or `NO`. On failure, the answer is always `MAYBE`. Based on the two possibilities success or failure, the semantics of the strategy operators is as follows.

The combinator '`;`' denotes sequential composition. Given two strategies s and s' together with a termination problem P, s`;`s' first tries to apply s to P. If this fails, then also s`;`s' fails, otherwise s' is applied to the resulting termination problem. So the strategy s`;`s' fails, whenever one of s and s' fails. The combinator '`|`' denotes choice. Different from sequential composition, the choice s`|`s' succeeds whenever at least one of s or s' succeeds. More precisely, given the strategy s`|`s', TTT2 first tries to apply s to P. If this succeeds, its result is the result of s`|`s', otherwise s' is applied to P. The combinator '`||`' is quite similar to the choice combinator and denotes parallel execution. That means given the strategy s`||`s', TTT2 runs s and s' in parallel on the termination problem P. As soon as at least one of s and s' succeeds, the resulting termination problem is returned. This can be seen as a kind of non-deterministic choice, since on simultaneous success of both s and s', it is more or less arbitrary whose result is taken.

Example A.1. Consider the following strategy:

```
dp;edg;sccs;(bounds -dp || (matrix -dp | kbo -af))
```

In order to prove termination of a TRS $\mathcal{R}$ using this strategy, TTT2 first computes the dependency pairs $\mathcal{P} = \mathsf{DP}(\mathcal{R})$ of $\mathcal{R}$ using the `dp` processor (thereby transforming the initially supplied TRS into a DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ with $\mathcal{G} = (\mathcal{P}, \mathcal{P} \times \mathcal{P})$). After that the estimated dependency graph and the SCCs of the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ are computed, resulting in a set of DP problems $\{(\mathcal{P}_1, \mathcal{R}, \mathcal{G}_1), \ldots, (\mathcal{P}_n, \mathcal{R}, \mathcal{G}_n)\}$ with $n \in \mathbb{N}$. Finally, to conclude that the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is finite and hence that the TRS $\mathcal{R}$ is terminating, TTT2 tries to prove finiteness of each DP problem $(\mathcal{P}_i, \mathcal{R}, \mathcal{G}_i)$ with $i \in \{1, \ldots, n\}$ by applying the match-bound technique and a combination of the matrix method and the Knuth-Bendix order in parallel. Here the substrategy `matrix -dp | kbo -af` first applies matrix interpretations to a given termination problem and on failure, applies the Knuth-Bendix order. Besides that the flag `-dp` specifies that e(-raise)-DP-boundedness should be proved in case of the `bounds` processor and weakly monotone interpretations should be used when the `matrix` processor is applied. The flag `-af` specifies that argument filterings should be considered when computing the Knuth-Bendix ordering.

Next we describe the iterators '`?`', '`+`', and '`*`'. The strategy s`?` tries to apply the strategy s to a termination problem P. On success its result is returned, otherwise P is returned unmodified. So s`?` applies s once or not at all to P and always succeeds. The iterators '`+`' and '`*`' are used to apply s recursively to P until P cannot be modified any more. The difference between '`+`' and '`*`' is that s`*` always succeeds whereas s`+` only succeeds if it can prove or disprove termination of P. In other words, s`*` is used to *simplify* problems, since it applies s until no further progress can be achieved and then returns the latest problem. In contrast '`+`' requires the proof attempt to be completed. This is particularly useful if two strategies should be combined of which the first aims to prove termination and the second one tends to prove non-termination.

Example A.2. We extend the strategy of the previous example by adding the iterators '`?`' and '`+`', as well as two new methods:

```
uncurry?;poly -ib 2 -ob 4*;
 dp;edg?;sccs?;(bounds -dp || (matrix -dp | kbo -af))+
```

To prove termination of a TRS $\mathcal{R}$, T$_\mathsf{T}$T$_2$ performs the following steps. At first uncurrying is applied. Since this method only works for applicative TRSs, the iterator '`?`' is added in order to avoid that the whole strategy fails if $\mathcal{R}$ is not an applicative system. After that polynomial interpretations with two input bits (coefficients) and four output bits (intermediate results) are used to simplify the given TRS. (Restricting the values for intermediate computations results in efficiency gains.) The iterator '`*`' ensures that a maximal number of rewrite rules is removed by applying the method as often as possible. Next, T$_\mathsf{T}$T$_2$ transforms the given termination problem into a DP problem. Afterwards the estimated dependency graph and the SCCs are computed. In contrast to the strategy of the previous example we have combined the processors `edg` and `sccs` with the iterator '`?`' because `edg` as well as `sccs` fail if they do not achieve any progress (`edg` fails if none of the arcs of the complete graph could be removed and `sccs` fails if the given termination problem could not be split into more than one problem). Finally, T$_\mathsf{T}$T$_2$ tries to prove finiteness of the given DP problems by applying the strategy `bounds -dp || (matrix -dp | kbo -af)` recursively.

At last we explain the specifiers '$\{s\}o$' and '`[`f`]`'. The strategy '$\{s\}o$' first tries to apply s to the given termination problem P. If this fails then also '$\{s\}o$' fails. Otherwise the modifier o is called. As input it gets the original termination problem P as well as the result of applying s to P (in the following called P'). Since o always succeeds, '$\{s\}o$' succeeds. Furthermore, as new termination problem the result of applying o to P' is returned. The specifier '`[`f`]`' denotes timed execution. Given a strategy s and a timeout f, s`[`f`]` tries to modify a given termination problem P for at most f seconds. If s does not succeed or fail within f seconds (wall clock time), s`[`f`]` fails. Otherwise s`[`f`]` returns the termination problem that remains after applying s to P. Hence it succeeds (fails) if s succeeds (fails).

Example A.3. To ensure that the strategy of the previous example is executed for at most 5 seconds we add the specifier '`[5]`'. In addition we limit the time spend by the match-bound technique to 1 second and combine it with usable rules. For the latter modification we use the operator '{ }' to emulate the DP processors obtained from Corollary 5.72. So the new strategy looks as follows:

```
(uncurry?;poly -ib 2 -ob 4*;dp;edg?;sccs?;
 ({ur?;bounds -dp[1]}restore || (matrix -dp | kbo -af))+)[5]
```

Using this strategy, $\mathsf{T_TT_2}$ has at most 5 seconds to prove the termination of a given TRS $\mathcal{R}$ and in each iteration 1 second is available to simplify termination problems using the match-bound technique. If the 5 seconds expire, the execution is aborted immediately. Of particular interest is the substrategy `{ur?;bounds -dp[1]}restore`. Assume that we apply this strategy to some problem P. First the usable rules of P are computed. Since the `ur` processor fails if all rewrite rules of P are usable we add the iterator `?` to avoid that the whole strategy fails if `ur` fails. Next the match-bound technique is applied for at most 1 second. If this processor succeeds the modifier `restore` is used to replace the usable rules by the original rewrite rules of P. To be able to do that, the processor `restore` gets as input the current problem as well as the old problem P. Note that if `bounds -dp[1]` fails then also `{ur?;bounds -dp[1]}restore` fails.

Specification and Configuration

In order to call $\mathsf{T_TT_2}$ with a certain strategy, the flag `--strategy` (or alternatively the short form `-s`) has to be set. For convenience it is possible to call $\mathsf{T_TT_2}$ without specifying any strategy. In this case a predefined strategy is used (for details execute `./ttt2 --help`). Note that the user is responsible for ensuring soundness of the strategy, for example, applying the processors in correct order.

Example A.4. To call $\mathsf{T_TT_2}$ with the strategy of Example A.3, the following command is used: `./ttt2 -s '(uncurry?; ...)[5]' <file>`. Alternatively, one could also remove the outermost time limit of the strategy and pass it as an argument to $\mathsf{T_TT_2}$. In that case the command looks as follows: `./ttt2 -s '(uncurry?; ...)' <file> 5`.

Since strategies can get quite complex (for instance, the strategy used in the 2009 edition of the international termination competition consists of about 100 lines), $\mathsf{T_TT_2}$ provides the opportunity to specify a configuration file. This allows to abbreviate and connect different strategies. By convention strategy abbreviations are written in capital letters. To tell $\mathsf{T_TT_2}$ which configuration file should be used, the flag `--conf` (or the short form `-c`) followed by the file name has to be set.

Example A.5. Consider the strategy of Example A.3. In order to call $\mathsf{T_TT_2}$ with this strategy we write a configuration file `ttt2.conf` containing the following lines:

```
PRE = uncurry?;poly -ib 2 -ob 4*
SUB = {ur;bounds -dp[1]}restore || (matrix -dp | kbo -af)
AUTO = (PRE;dp?;edg?;sccs?;(SUB)+)[5]
```

It is important to note that abbreviations are not implicitly surrounded by parentheses since this allows more freedom in abbreviating expressions. To inform $\mathsf{T_TT_2}$ that the strategy `AUTO` of the configuration file `ttt2.conf` should be used to prove the termination of a given TRS the following flags have to be declared: `./ttt2 -c ttt2.conf -s AUTO <file>`.

Used Strategies

In the following we shortly introduce the strategies that have been used to conduct the experiments in Sections 7.1 and 7.2. To obtain the experimental data shown in Tables 7.1 and 7.2, $\mathsf{T_TT_2}$ has been configurated using the following strategies:

	explicit
t	`bounds -e top -rc explicit`
r	`bounds -e roof -rc explicit`
rm	`bounds -e match -rc explicit`

	implicit
t	`bounds -e top -rc implicit`
r	`bounds -e roof -rc implicit`
rm	`bounds -e match -rc implicit`

To compute quasi-compatible tree automata instead of compatible tree automata we equipped each instance of the `bounds` processor with the additional flag `-qc`. Similarly, to incorporate right-hand sides of forward closures the flag `-rfc` has been appended to each strategy. To conduct the experiments for Tables 7.3 and 7.4 we used the basic strategy

```
dp;edg?;sccs?;(sc[5] | matrix -dp -dim 1 -ib 1 -ob 2[5] | s)*
```

where s has been chosen as follows:

	without ur	with ur
sp	`fail`	
spb	`bounds`	`{ur -ce?;bounds}restore`
spd	`bounds -dp`	`{ur -ce?;bounds -dp}restore`

In addition we added the flags `-qc` and `-rfc` to each `bounds` processor in order to compute quasi-compatible tree automata or to ensure that right-hand sides of forward closures are in effect.

To produce the experimental data presented in Subsection 7.2.1, $\mathsf{T_TT_2}$ has been called with the strategy `dp;`s`;sccs?` where s has been replaced either by a single graph processor or a combination of graph processors. In case of $*$, s has been defined as

```
{ur -ce?;tdg?;edg[0.5]?;cdg[0.5]?;cdg -rfc[0.5]?}restore
```

if usable rules should be computed and

```
tdg?;edg[0.5]?;cdg[0.5]?;cdg -rfc[0.5]?
```

otherwise. The experimental data of all other graph processors have been obtained by replacing s according to the following table:

	without `ur`	with `ur`
`t`	`tdg?`	`{ur -ce?;tdg?}restore`
`e`	`edg?`	`{ur -ce?;edg?}restore`
`s`	`adg -a strong?`	`{ur -ce?;adg -a strong?}restore`
`nv`	`adg -a newars?`	`{ur -ce?;adg -a newars?}restore`
`g`	`adg -a growing?`	`{ur -ce?;adg -a growing?}restore`
`c`	`cdg?`	`{ur -ce?;cdg?}restore`
`r`	`cdg -rfc?`	`{ur -ce?;cdg -rfc?}restore`

In addition, if polynomial interpretations were used, the processor

```
matrix -dp -dim 1 -ib 1 -ob 2*
```

has been appended to the strategy. So instead of `dp;`s`;sccs?` the strategy `dp;`s`;sccs?;matrix -dp -dim 1 -ib 1 -ob 2*` has been used. To conduct the experiments in Subsection 7.2.2 we used the same strategies as for the experiments in Subsection 7.2.1, except that the processors `edg` and `cdg` have been additionally equipped with the flag `-i`. So in case of the innermost DP processors `e`, `c`, and `r` the substrategies

	without `ur`	with `ur`
`e`	`edg -i?`	`{ur?;edg -i?}restore`
`c`	`cdg -i?`	`{ur?;cdg -i?}restore`
`r`	`cdg -i -rfc?`	`{ur?;cdg -i -rfc?}restore`

have been applied. Likewise, for $*$ the substrategies

```
{ur?;tdg?;edg -i[0.5]?;cdg -i[0.5]?;cdg -i -rfc[0.5]?}restore
```

if usable rules should be computed and

```
tdg?;edg -i[0.5]?;cdg -i[0.5]?;cdg -i -rfc[0.5]?
```

otherwise have been used. Note that the `ur` processor has been called without the flag `-ce` because for innermost termination it is not necessary to add any projection rules. So basically it would be also possible to drop the modifier `restore` since it is not necessary to replace the usable rules by the original ones after the graph has been computed. We did not do that to ensure that the results are comparable with the ones where usable rules are not used (without replacing the usable rules by the original rules, the `matrix` processor would be more powerful since less rules have to be oriented).

A.2 The Complexity Tool CaT

The tool CaT (short form for *Complexity and Termination*) is a derivative of TTT2 especially configurated to prove polynomial upper bounds on the derivational complexity and runtime complexity of TRSs. Similar as the tool TTT2, CaT is open source, published under terms of the GNU Lesser General Public License. Precompiled binaries, sources, and documentation of CaT are available at the following web-site:

```
http://cl-informatik.uibk.ac.at/software/cat/
```

In the remainder of this section we refer to version 1.5 of the tool.

A.2.1 Design and Execution

The aim of CaT is just to demonstrate how helpful it is to start from the basis of a well-designed termination prover; the additional implementation effort took a single day. Since CaT and TTT2 share most of the code, we recently decided to completely integrate the additional code fragments of CaT into TTT2. As a result, the source code of CaT is nearly identical to the one of TTT2 (there are only some slight differences in some of the makefiles). So CaT can be seen as a doubleganger of TTT2 especially aimed for complexity analysis. The only difference between the tools is that in case of CaT the so called complexity mode is per default enabled.

In order to run CaT from the command line, the user can either download the source code from the CaT web page and install it following the installation guidelines or alternatively download the binary of the latest version of CaT. After a successful installation, CaT can be started via the command

```
./cat [options] <file> [timeout]
```

where `[options]` denotes a list of command line options, `<file>` specifies the name of the file containing the TRS of which termination should be proved, and `[timeout]`—a floating point number—defines the time limit for proving termination of the given TRS. The timeout is optional. To get a complete list of the command line options of CaT execute the command `./cat --help`.

The most important option for complexity analysis is the flag `--complexity` (or its short form `-cp`). Via this flag the user can control if either the derivational or runtime complexity of a given TRS should be computed. To do this the flag `-cp` has to be equipped with one of the following options:

- `DC` specifies that the derivational complexity of the given TRS should be computed, even when the problem file states something different.
- `PC` defines that the complexity category, induced by the given problem file, should be chosen. That means, if the file specifies as start terms the set of all constructor-based terms then the runtime complexity is computed. Otherwise, the derivational complexity of the given TRS is estimated.
- `RC` ensures that CaT estimates the runtime complexity of the given TRS. Similar as for `DC`, any information in the problem file regarding complexity analysis is ignored.

A.2.2 The Strategy Language

As mentioned in the introduction, there is basically no difference between CaT and TTT2, except the name. As a result the strategy language of CaT is identical to the one of $\mathsf{T_TT_2}$. However, to generate a strategy that can be used to prove a polynomial complexity of a given TRS the user has to ensure that the used methods are properly configurated. Detailed information how this can be done can be inferred from the documentation.

Computing Complexity Bounds

To compute the complexity of a given termination problem P using some strategy s, CaT proceeds as follows. First it applies s to P. During this step a internal proof object is created which contains basic information about the applied processors, the intermediate termination problems, etc. Afterwards this internal proof object is analyzed in order to compute the complexity of the input problem P. Possible answers of the prover are: `YES(`c_l`,`c_u`)`, `NO`, and `MAYBE`. Here `YES(`c_l`,`c_u`)` means that P is terminating and that the exact complexity of P is somewhere between the lower bound c_l and the upper bound c_u, `NO` indicates a successful non-termination proof, and `MAYBE` refers to the case when termination could neither be proved nor disproved. Possible values for c_l and c_u are `?`, `O(1)`, `O(n`^{i}`)` for some $i \in \mathbb{N}$, and `POLY`. Here `?` indicates that the complexity of P is unknown and `POLY` means that the complexity of P is polynomial (the exact bound is unknown). So far CaT can only prove upper complexity bounds. So a successful complexity proof of CaT is always indicated by the answer `YES(?,`c_u`)`. If CaT returns the answer `YES(?,?)` then some method has been used which is unknown to infer polynomial upper complexity bounds. In such a case the used strategy should be adapted.

As indicated in the previous subsection, in general we differentiate between the derivational complexity and the runtime complexity of TRSs. Per default, CaT automatically chooses the complexity category induced by the problem file. That means, if the input problem specifies as start terms the set of all terms, the derivational complexity of the given TRS is computed. Otherwise, if the set of start terms is set to the set of all constructor-based terms, the runtime complexity of the given problem is estimated.

Used Strategies

In the following we shortly discuss the strategies that have been used to conduct the experiments in Section 7.3. The information shown in Table 7.11 has been obtained by using the strategy

```
cp;(matrix -dim 1 -ib 1 -ob 2 -cp -triangle -strict[5])*;
 (matrix -dim 2 -ib 1 -ob 2 -cp -triangle ||
  (sleep -t 1?;matrix -dim 3 -ib 1 -ob 2 -cp -triangle))*
```

for the column labeled with `pm`, the strategy

```
bounds[5]?;cp;
 (matrix -dim 1 -ib 1 -ob 2 -cp -triangle -strict[5])*;
 (matrix -dim 2 -ib 1 -ob 2 -cp -triangle ||
  (sleep -t 1?;matrix -dim 3 -ib 1 -ob 2 -cp -triangle))*
```

when match-bounds are in effect (column `bpm`), and

```
bounds[5]?;cp;
 (matrix -dim 1 -ib 1 -ob 2 -cp -triangle -strict[5])*;
 (bounds -cp[5])*;(matrix -dim 2 -ib 1 -ob 2 -cp -triangle ||
  (sleep -t 1?;matrix -dim 3 -ib 1 -ob 2 -cp -triangle))*
```

when the CP processors based on match(-raise)-RT-bounds are used (column bprm). The main difference between the latter two strategies is that in the third one an additional instance of the `bounds` processor has been added which is aimed to prove match(-raise)-RT-boundedness and hence linear complexity bounds of single rewrite rules. So in the third strategy the `bounds` processor can assist the other processors to prove linear, quadratic, or even higher complexity bounds. The reason why we did not add a second instance of the `bounds` processor in the second strategy is that the ordinary match-bound technique does not allow us to prove the complexity of single rewrite rules. So it would not make sense to add an additional instance of the `bounds` processor because it would behave in the exact same manner as the first instance.

In case of runtime complexity (see Table 7.12) we used the same strategies as for derivational complexity except that all instances of the `bounds` processor have been equipped with the additional flag `-l constructor`. Furthermore, to construct quasi-compatible tree automata instead of compatible tree automata we equipped each `bounds` processor with the additional flag `-qc`.

Appendix B

Supplementary Proofs

In this chapter we present the proofs of Lemmata 5.10, 5.26, and 5.40. To prove these statements we mark function symbols of terms as active and inactive either to trace the propagation of heights as in case of Lemma 5.10 or the application of rewrite rules as in case of Lemmata 5.26 and 5.40. The idea to consider active and inactive areas of terms occurring in derivations originates from [7]. Below we recall the most important notions and results.

Most of the information presented in this appendix appeared already in the journal paper [43]. However the prove of Lemma 5.26 is slightly reformulated such that it can be reused to prove Lemma 5.40.

B.1 Preliminaries

For a signature $\mathcal{F}$, $\overline{\mathcal{F}}$ denotes the set $\{\overline{f} \mid f \in \mathcal{F}\}$ where $\overline{f}$ is a fresh function symbol with the same arity as f. The function symbols in $\overline{\mathcal{F}}$ are called *active* whereas those in $\mathcal{F}$ are called *inactive*. The mappings $\mathsf{label}\colon \mathcal{F} \to \overline{\mathcal{F}}$ and $\mathsf{unlabel}\colon \overline{\mathcal{F}} \to \mathcal{F}$ are defined as $\mathsf{label}(f) = \overline{f}$ and $\mathsf{unlabel}(\overline{f}) = f$. They are extended to terms and sets of terms in the obvious way. A term $s \in \mathcal{T}(\overline{\mathcal{F}} \cup \mathcal{F}, \mathcal{V})$ is called *inactive* if $s = \mathsf{unlabel}(s)$. For a term $s \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ the set $\mathsf{mark}(s)$ consists of all terms t which can be divided into a context $C \in \mathcal{T}(\overline{\mathcal{F}} \cup \{\square\}, \mathcal{V})$ and terms $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that $t = C[t_1, \ldots, t_n]$ and $\mathsf{unlabel}(t) = s$. For two terms $s, t \in \mathcal{T}(\overline{\mathcal{F}} \cup \mathcal{F}, \mathcal{V})$ with $\mathsf{unlabel}(s) = \mathsf{unlabel}(t)$ we write $s \uparrow t$ for the term u that is uniquely determined by the following two conditions: $\mathsf{unlabel}(u) = \mathsf{unlabel}(s)$ and for each position $p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(u)$ we have $u(p) \in \overline{\mathcal{F}}$ if and only if $s(p) \in \overline{\mathcal{F}}$ or $t(p) \in \overline{\mathcal{F}}$. We extend this notion to $\uparrow S$ for finite non-empty sets $S \subset \mathcal{T}(\overline{\mathcal{F}} \cup \mathcal{F}, \mathcal{V})$ consisting of terms that have the same unlabeled image.

Definition B.1. Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$. The TRS $\overline{\mathcal{R}}$ over the signature $\overline{\mathcal{F}} \cup \mathcal{F}$ consists of all rewrite rules $l \to r$ for which there exists a rewrite rule $l' \to \mathsf{unlabel}(r) \in \mathcal{R}$ such that $l \in \mathsf{mark}(l')$ and $r \in \mathcal{T}(\mathcal{G}, \mathcal{V})$ where $\mathcal{G} = \mathcal{F}$ if $l(\epsilon) \in \mathcal{F}$ and $\mathcal{G} = \overline{\mathcal{F}}$ otherwise.

Example B.2. Let $\mathcal{R}$ be the TRS consisting of the rewrite rules

$$\mathsf{f}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{g}(x, \mathsf{p}(y))) \qquad \mathsf{g}(\mathsf{p}(x), \mathsf{s}(y)) \to y \qquad \mathsf{g}(x, x) \to \mathsf{f}(\mathsf{s}(x), x)$$

over the signature $\mathcal{F} = \{\mathsf{f}, \mathsf{g}, \mathsf{p}, \mathsf{s}\}$. We have $\overline{\mathcal{F}} = \{\bar{\mathsf{f}}, \bar{\mathsf{g}}, \bar{\mathsf{p}}, \bar{\mathsf{s}}\}$. The set $\mathsf{mark}(t)$ with $t = \mathsf{g}(\mathsf{p}(x), \mathsf{s}(y))$ consists, besides t, of the terms $\bar{\mathsf{g}}(\mathsf{p}(x), \mathsf{s}(y))$, $\bar{\mathsf{g}}(\bar{\mathsf{p}}(x), \mathsf{s}(y))$,

$\overline{\mathsf{g}}(\mathsf{p}(x), \overline{\mathsf{s}}(y))$, and $\overline{\mathsf{g}}(\overline{\mathsf{p}}(x), \overline{\mathsf{s}}(y))$. Furthermore, the TRS $\overline{\mathcal{R}}$ contains the following rewrite rules:

$$\begin{array}{lll}
\mathsf{f}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{g}(x, \mathsf{p}(y))) & \mathsf{g}(\mathsf{p}(x), \mathsf{s}(y)) \to y & \mathsf{g}(x, x) \to \mathsf{f}(\mathsf{s}(x), x) \\
\overline{\mathsf{f}}(\mathsf{s}(x), y) \to \overline{\mathsf{s}}(\overline{\mathsf{g}}(x, \overline{\mathsf{p}}(y))) & \overline{\mathsf{g}}(\mathsf{p}(x), \mathsf{s}(y)) \to y & \overline{\mathsf{g}}(x, x) \to \overline{\mathsf{f}}(\overline{\mathsf{s}}(x), x) \\
\overline{\mathsf{f}}(\overline{\mathsf{s}}(x), y) \to \overline{\mathsf{s}}(\overline{\mathsf{g}}(x, \overline{\mathsf{p}}(y))) & \overline{\mathsf{g}}(\overline{\mathsf{p}}(x), \mathsf{s}(y)) \to y & \\
 & \overline{\mathsf{g}}(\mathsf{p}(x), \overline{\mathsf{s}}(y)) \to y & \\
 & \overline{\mathsf{g}}(\overline{\mathsf{p}}(x), \overline{\mathsf{s}}(y)) \to y &
\end{array}$$

Definition B.3. Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$. We define the relation $\rightarrowtriangle$ on $\mathcal{T}(\mathcal{F} \cup \overline{\mathcal{F}}, \mathcal{V})$ as follows: $s \rightarrowtriangle_{\mathcal{R}} t$ if and only if there exist a rewrite rule $l \to r \in \overline{\mathcal{R}}$, a position $p \in \mathcal{P}\mathsf{os}(s)$, a context C, and terms $s_1, \ldots, s_n$ such that $l = C[x_1, \ldots, x_n]$ with all variables displayed, $s|_p = C[s_1, \ldots, s_n]$, $\mathsf{unlabel}(s_i) = \mathsf{unlabel}(s_j)$ whenever $x_i = x_j$, and $t = s[r\sigma]_p$. Here the substitution σ is defined as follows:

$$\sigma(x) = \begin{cases} \Uparrow\{s_i \mid x_i = x \text{ with } i \in \{1, \ldots, n\}\} & \text{if } x \in \{x_1, \ldots, x_n\} \\ x & \text{otherwise} \end{cases}$$

An immediate consequence of the next lemma is that every rewrite sequence caused by the TRS $\mathcal{R}$ can be lifted to a rewrite sequence in $\overline{\mathcal{R}}$.

Lemma B.4. *Let $\mathcal{R}$ be a TRS. If $s \to_{\mathcal{R}} t$ then for all terms $s' \in \mathsf{mark}(s)$ there exists a term $t' \in \mathsf{mark}(t)$ such that $s' \rightarrowtriangle_{\mathcal{R}} t'$.* □

Definition B.5. Let $\mathcal{R}$ be a TRS over the signature $\mathcal{F}$, $l \to r \in \overline{\mathcal{R}}$ a rewrite rule, t a term, $p \in \mathcal{P}\mathsf{os}(t)$ a position, and σ a substitution. The rewrite step $t[l\sigma]_p \rightarrowtriangle t[r\sigma]_p$ is said to be *active* if $t(p) \in \overline{\mathcal{F}}$ and *inactive* if $t(p) \in \mathcal{F}$. We use $\overset{\mathsf{a}}{\rightarrowtriangle}$ to denote active steps and $\overset{\mathsf{i}}{\rightarrowtriangle}$ to denote inactive steps. An active rewrite step $t[l\sigma]_p \overset{\mathsf{a}}{\rightarrowtriangle} t[r\sigma]_p$ is said to be *strongly active* if $\mathcal{F}\mathsf{un}(l) \subseteq \overline{\mathcal{F}}$.

Example B.6. With respect to the TRS $\mathcal{R}$ of the previous example, the term $\overline{\mathsf{f}}(\mathsf{s}(\mathsf{p}(x)), \mathsf{g}(\mathsf{p}(y), \mathsf{s}(x)))$ admits the following rewrite sequence:

$$\begin{aligned}
\overline{\mathsf{f}}(\mathsf{s}(\mathsf{p}(x)), \mathsf{g}(\mathsf{p}(y), \mathsf{s}(x))) &\overset{\mathsf{i}}{\rightarrowtriangle}_{\mathcal{R}} \overline{\mathsf{f}}(\mathsf{s}(\mathsf{p}(x)), x) \\
&\overset{\mathsf{a}}{\rightarrowtriangle}_{\mathcal{R}} \overline{\mathsf{s}}(\overline{\mathsf{g}}(\mathsf{p}(x), \overline{\mathsf{p}}(x))) \\
&\overset{\mathsf{a}}{\rightarrowtriangle}_{\mathcal{R}} \overline{\mathsf{s}}(\overline{\mathsf{f}}(\overline{\mathsf{s}}(\overline{\mathsf{p}}(x)), \overline{\mathsf{p}}(x)))
\end{aligned}$$

Note that the second active rewrite step is strongly active.

A second important property of $\overline{\mathcal{R}}$ is that active function symbols always stay above inactive function symbols.

Lemma B.7. *Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$ and $s \in \mathcal{T}(\overline{\mathcal{F}} \cup \mathcal{F}, \mathcal{V})$ a term such that $s \in \mathsf{mark}(\mathsf{unlabel}(s))$. If $s \rightarrowtriangle^*_{\mathcal{R}} t$ then $t \in \mathsf{mark}(\mathsf{unlabel}(t))$.* □

The proofs of Lemmata 5.10, 5.26, and 5.40 are based on the observation that from some point on in each minimal rewrite sequence only strongly active steps are applied. Below we prove the correctness of this observation. Minimal terms in Lemma B.8 are terms that have the property that all proper subterms are terminating.

Lemma B.8. *Let $\mathcal{R}$ be a non-duplicating TRS over some signature $\mathcal{F}$ and $s \in \mathcal{T}(\overline{\mathcal{F}} \cup \mathcal{F}, \mathcal{V})$ a minimal term such that the root symbol of s is active and all other function symbols are inactive. If s starts a infinite rewrite sequence of the form*

$$s = s_1 \rightarrowtriangle_{\mathcal{R}} s_2 \rightarrowtriangle_{\mathcal{R}} s_3 \rightarrowtriangle_{\mathcal{R}} s_4 \rightarrowtriangle_{\mathcal{R}} \cdots$$

then there exists an $i \geqslant 1$ such that all rewrite steps in the rewrite sequence starting from s_i are strongly active.

Proof. The proof of this lemma is based on the following observations:

- active $\overset{\mathsf{a}}{\rightarrowtriangle}_{\mathcal{R}}$-steps cannot increase the number of inactive symbols because $\mathcal{R}$ is non-duplicating,
- proper subterms of the term s are terminating due to the minimality assumption,
- maximal inactive subterms of s_j for $j \geqslant 1$ can be traced back to inactive subterms of s.

The first two observations hold trivially. To show the correctness of the last one, we prove the following claim:

> If $t \rightarrowtriangle_{\mathcal{R}}^* u$ with $t, u \in \mathcal{T}(\overline{\mathcal{F}} \cup \mathcal{F}, \mathcal{V})$ and $t \in \mathsf{mark}(\mathsf{unlabel}(t))$, then for each inactive subterm u' of u there is an inactive subterm t' of t and a context C such that $t' \overset{\mathsf{i}}{\rightarrowtriangle}_{\mathcal{R}}^* C[u']$.

We prove the claim by induction on the length of the derivation. The base case is trivial. Assume now that $t \rightarrowtriangle_{\mathcal{R}}^+ u$. Then there are a position p, a substitution σ, and a rewrite rule $l \to r \in \overline{\mathcal{R}}$ such that $t \rightarrowtriangle_{\mathcal{R}}^* u[l\sigma]_p \rightarrowtriangle u[r\sigma]_p = u$. Let u' be an inactive subterm of u at some position $q \parallel p$. Then $u' = (u[l\sigma]_p)|_q$. Due to the induction hypothesis we know that there exists an inactive subterm t' of t and a context C such that $t' \overset{\mathsf{i}}{\rightarrowtriangle}_{\mathcal{R}}^* C[u']$. Assume now that u' is an inactive subterm of u at some position q such that either $q < p$ or $q \geqslant p$. We distinguish between these two cases.

- If $q < p$ then $u[l\sigma]_p \overset{\mathsf{i}}{\rightarrowtriangle}_{l \to r} u$. Let $v = (u[l\sigma]_p)|_q$. Obviously, v is an inactive subterm of $u[l\sigma]_p$ and $v \overset{\mathsf{i}}{\rightarrowtriangle}_{l \to r} u'$. The induction hypothesis yields an inactive subterm t' of t and a context C such that $t' \overset{\mathsf{i}}{\rightarrowtriangle}_{\mathcal{R}}^* C[v]$. Hence $t' \overset{\mathsf{i}}{\rightarrowtriangle}_{\mathcal{R}}^* C[v] \overset{\mathsf{i}}{\rightarrowtriangle}_{l \to r} C[u']$.
- If $q \geqslant p$ then either $u[l\sigma]_p \overset{\mathsf{i}}{\rightarrowtriangle}_{l \to r} u$ or $u[l\sigma]_p \overset{\mathsf{a}}{\rightarrowtriangle}_{l \to r} u$. In the former case we have $l\sigma \overset{\mathsf{i}}{\rightarrowtriangle}_{l \to r} D[u']$ for some context D. The induction hypothesis yields an inactive subterm t' of t and a context C such that $t' \overset{\mathsf{i}}{\rightarrowtriangle}_{\mathcal{R}}^* C[l\sigma]$. Hence $t' \overset{\mathsf{i}}{\rightarrowtriangle}_{\mathcal{R}}^* C[l\sigma] \overset{\mathsf{i}}{\rightarrowtriangle}_{l \to r} C[D[u']]$. Next suppose that $u[l\sigma]_p \overset{\mathsf{a}}{\rightarrowtriangle}_{l \to r} u$. Since all function symbols of r are active, we conclude that u' is a subterm of $x\sigma$ for some variable $x \in \mathcal{V}\mathsf{ar}(r)$. Hence u' is an inactive subterm of $u[l\sigma]_p$. The induction hypothesis yields an inactive subterm t' of t and a context C such that $t' \overset{\mathsf{i}}{\rightarrowtriangle}_{\mathcal{R}}^* C[u']$.

This concludes the proof of the claim.

Now, from the above observations it follows that the infinite sequence starting from s contains only finitely many inactive $\overset{\mathsf{i}}{\rightarrowtriangle}_{\mathcal{R}}$-steps. Hence a tail of the sequence consists entirely of $\overset{\mathsf{a}}{\rightarrowtriangle}_{\mathcal{R}}$-steps. Steps in this tail that are not strongly active consume at least one inactive symbol whereas the strongly active steps do not increase the number of inactive symbols. Hence from some point on only strongly active steps are applied. This completes the proof of the lemma. $\square$

The following example shows that the previous lemma does not hold for duplicating TRSs. Hence it cannot be used for instance to prove soundness of $\mathsf{roof\text{-}DP}(\mathcal{P}, s \to t, \mathcal{R})$.

Example B.9. Consider the TRS $\mathcal{R}$ consisting of the rewrite rules $\mathsf{a} \to \mathsf{b}$ and $\mathsf{f}(\mathsf{a}, \mathsf{b}, x) \to \mathsf{f}(x, x, x)$. The term $\bar{\mathsf{f}}(\mathsf{a}, \mathsf{b}, \mathsf{a})$ admits the rewrite sequence

$$\bar{\mathsf{f}}(\mathsf{a}, \mathsf{b}, \mathsf{a}) \overset{\mathsf{a}}{\rightarrowtriangle}_{\mathcal{R}} \bar{\mathsf{f}}(\mathsf{a}, \mathsf{a}, \mathsf{a}) \overset{\mathsf{i}}{\rightarrowtriangle}_{\mathcal{R}} \bar{\mathsf{f}}(\mathsf{a}, \mathsf{b}, \mathsf{a}) \overset{\mathsf{a}}{\rightarrowtriangle}_{\mathcal{R}} \cdots$$

Since this sequence does not contain any strongly active rewrite steps, it is obvious that the previous lemma does not hold.

B.2 Soundness of Match(-Raise)-DP-Bounds

By using the rewrite relation $\rightarrowtriangle_{\mathsf{match\text{-}DP}(\mathcal{P}, s \to t, \mathcal{R})}$ we are now ready to prove that no restriction of the TRS $\mathsf{match\text{-}DP}(\mathcal{P}, s \to t, \mathcal{R})$ to a finite signature admits minimal rewrite sequences with infinitely many $\xrightarrow{\epsilon}_{\mathsf{match}(s \to t)}$-steps.

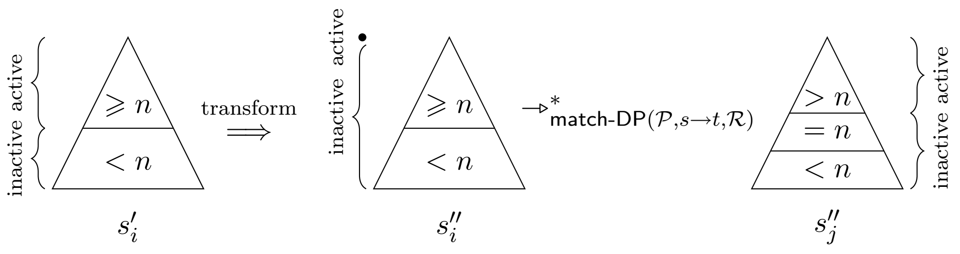
Proof of Lemma 5.10. Assume to the contrary that there is a minimal rewrite sequence of the form

$$s_1 \xrightarrow{\epsilon}_{\mathsf{match}(s \to t)} t_1 \to^*_{\mathsf{match\text{-}DP}(\mathcal{P}, s \to t, \mathcal{R})} s_2 \xrightarrow{\epsilon}_{\mathsf{match}(s \to t)} t_2 \to^*_{\mathsf{match\text{-}DP}(\mathcal{P}, s \to t, \mathcal{R})} \cdots$$

where we assume without loss of generality that $s_1 \in \mathcal{T}(\mathcal{F}_{\{0\}}, \mathcal{V})$. Here $\mathcal{F}$ denotes the signature of $\mathcal{P} \cup \mathcal{R}$. To trace the propagation of heights in this sequence we switch from $\to$ to $\rightarrowtriangle$. To simplify the representation we assume that for terms $s_i^{\prime \cdots \prime}$ we have $\mathsf{unlabel}(s_i^{\prime \cdots \prime}) = s_i$. Moreover, the infinite rewrite sequence starting from $s_i^{\prime \cdots \prime}$ is projected onto the above sequence from s_i by applying the function $\mathsf{unlabel}$. The terms $s_i^{\prime \cdots \prime}$ will be constructed as we go along. To prove the lemma, we first prove the following claim (illustrated in Figure B.1):

> Let s_i' be a term such that all rewrite steps in the infinite rewrite sequence starting from s_i' are strongly active and the height of all active symbols in s_i' is at least n. Further let s_i'' be the term obtained from s_i' by labeling all function symbols as inactive except the root symbol. Then there is a term s_j'' with $j \geqslant i$ such that all rewrite steps of the infinite sequence starting at s_j'' are strongly active and the height of every active function symbol in s_j'' is at least $n + 1$.

Figure B.1: The claim in the proof of Lemma 5.10



Lemma B.8 yields a term s''_j with $j \geqslant i$ such that all rewrite steps of the rewrite sequence starting from s''_j are strongly active. Because all rewrite steps in the infinite rewrite sequence starting from s'_i are strongly active, we know that whenever

$$s'_i \xrightarrow{\mathsf{a}}{}^*_{\mathsf{match\text{-}DP}(\mathcal{P},s\to t,\mathcal{R})} u = u[l\sigma]_p \xrightarrow{\mathsf{a}}_{\mathsf{match\text{-}DP}(\mathcal{P},s\to t,\mathcal{R})} u[r\sigma]_p$$

for some term u, rewrite rule $l \to r$, position p, and substitution σ, the minimal height of function symbols in l is at least n. Since the rewrite sequence starting from s'_i is equivalent to the rewrite sequence starting at the term s''_i after unlabeling, this property holds also for s''_i. Since the rewrite step $s''_i \rightarrowtriangle_{\mathsf{match}(s\to t)} t''_i$ is active, we know that the height of all active function symbols in t''_i is at least $n+1$. Hence, whenever

$$t''_i \rightarrowtriangle^*_{\mathsf{match\text{-}DP}(\mathcal{P},s\to t,\mathcal{R})} u = u[l\sigma]_p \xrightarrow{\mathsf{a}}_{\mathsf{match\text{-}DP}(\mathcal{P},s\to t,\mathcal{R})} u[r\sigma]_p$$

the height of the root symbol of l is at least $n+1$. Together with the fact that the minimal height of the function symbols in the redex pattern of an active rewrite step is at least n, we can conclude from Definition 5.3 that the height of each active function symbol in s''_j must be at least $n+1$. This completes the proof of the claim.

Now let s'_1 be the term obtained from s_1 by marking the root symbol as active. Lemma B.8 yields a term s'_{i_1} with $i_1 \geqslant 1$ such that $s'_1 \rightarrowtriangle^*_{\mathsf{match\text{-}DP}(\mathcal{P},s\to t,\mathcal{R})} s'_{i_1}$ and all rewrite steps in the rewrite sequence starting from s'_{i_1} are strongly active. Since $s'_1 \xrightarrow{\mathsf{a}}_{\mathsf{match}(s\to t)} t'_1$, we know that the height of every active function symbol in t'_1 is 1. It follows that the height of each active function symbol in s'_{i_1} is at least 1. Let s''_{i_1} be the term obtained from s'_{i_1} by inactivating all function symbols below the root. Applying the claim yields a term s''_{i_2} with $i_2 \geqslant i_1$ such that all rewrite steps in the infinite sequence starting from s''_{i_2} are strongly active and the height of all active function symbols in s''_{i_2} is at least 2. Repeating this argumentation produces increasingly greater heights. As soon as we reach height $c+1$ we obtain a contradiction with the assumptions of Lemma 5.10. $\square$

B.3 Soundness of Forward Closures

In this section we present the proofs of Lemmata 5.26 and 5.40. First we prove the correctness of Lemma 5.26. Similar as in the proof of Lemma 5.10, we use the rewrite relation $\rightarrowtriangle$ to obtain detailed information about derivations.

Proof of Lemma 5.26. We prove the statement of the lemma by showing a slightly stronger result:

> Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs and $s \to t \in \mathcal{P}$ a rewrite rule. If $\mathcal{P}$ and $\mathcal{R}$ are right-linear then the TRS $\mathcal{P} \cup \mathcal{R}$ admits a minimal rewrite sequence of the form $s_1 \xrightarrow{\epsilon}_{\alpha_1} t_1 \to^*_{\mathcal{R}} s_2 \xrightarrow{\epsilon}_{\alpha_2} t_2 \to^*_{\mathcal{R}} \cdots$ with infinitely many $\xrightarrow{\epsilon}_{s\to t}$-steps if and only if there is a term $w \in \mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R})$ starting a minimal rewrite sequence $w \xrightarrow{\epsilon}_{\alpha_i} \cdot \to^*_{\mathcal{R}} \cdot \xrightarrow{\epsilon}_{\alpha_{i+1}} \cdot \to^*_{\mathcal{R}} \cdots$ for some $i \geqslant 1$.

The if direction of the claim holds trivially. To prove the only-if direction, let

$$s_1 \xrightarrow{\epsilon}_{\alpha_1} t_1 \to^*_{\mathcal{R}} s_2 \xrightarrow{\epsilon}_{\alpha_2} t_2 \to^*_{\mathcal{R}} \cdots$$

be a minimal rewrite sequence with infinitely many $\xrightarrow{\epsilon}_{s\to t}$-steps. Without loss of generality we assume that $\alpha_1 = s \to t$. To trace the application of rewrite rules in this sequence we switch from $\to$ to $\rightarrowtriangle$. Let s'_1 be the term obtained from s_1 by marking the root symbol as active. Lemma B.8 yields an $i \geqslant 1$ such that all rewrite steps in the rewrite sequence starting from s'_i are strongly active. Let $l_1 \to r_1, \ldots, l_n \to r_n$ be (fresh variants of) rewrite rules in $\mathcal{P} \cup \mathcal{R}$ such that

$$s'_1 \overset{\mathsf{a}}{\rightarrowtriangle}_{s\to t} t'_1 \overset{\mathsf{i}}{\rightarrowtriangle}^*_{\mathcal{R}} u_1 \overset{\mathsf{a}}{\rightarrowtriangle}_{l_1\to r_1} v_1 \overset{\mathsf{i}}{\rightarrowtriangle}^*_{\mathcal{R}} u_2 \overset{\mathsf{a}}{\rightarrowtriangle}_{l_2\to r_2} v_2 \overset{\mathsf{i}}{\rightarrowtriangle}^*_{\mathcal{R}} \cdots \overset{\mathsf{a}}{\rightarrowtriangle}_{l_n\to r_n} v_n \overset{\mathsf{i}}{\rightarrowtriangle}^*_{\mathcal{R}} s'_i$$

with all active steps displayed and let $p_1, \ldots, p_n$ be the positions at which the rewrite rules $l_1 \to r_1, \ldots, l_n \to r_n$ are applied. To prove the lemma we first show that we can characterize the active region of s'_i with the help of the definition of the right-hand sides of forward closures. Let $v_0 = t'_1$. We define linear terms $w_j \in \mathcal{T}(\overline{\mathcal{F}}, \mathcal{V})$ and substitutions $\tau_j \colon \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ for $j \in \{0, \ldots, n\}$ such that the following properties hold: (1) $v_j = w_j\tau_j$ and (2) $\mathsf{unlabel}(w_j) \in \mathsf{RFC}_t(\mathcal{P} \cup \mathcal{R})$.

We perform induction on j. First consider $j = 0$. Define $w_0 = \mathsf{label}(t)$. Since $\mathsf{unlabel}(w_0) = t$, property (2) holds trivially. Since $\mathcal{P}$ is right-linear, w_0 is linear. Obviously, $w_0 \in \mathcal{T}(\overline{\mathcal{F}}, \mathcal{V})$. Since $v_0 = t'_1$ is an instance of $\mathsf{label}(t)$, there exists a substitution τ_0 such that $v_0 = w_0\tau_0$. We may assume that τ_0 maps variables in $\mathcal{V}$ to terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$ as there are no other active symbols in v_0 besides the ones in $\mathsf{label}(t)$. Hence property (1) also holds. Now let $j \in \{1, \ldots, n\}$. Since all steps that take place between v_{j-1} and u_j are inactive, we infer that the active part part of u_j is identical to the active part of v_{j-1}. Since the active rewrite step $u_j \overset{\mathsf{a}}{\rightarrowtriangle}_{l_j\to r_j} v_j$ requires that the root symbol of the contracted redex is active we know that p_j is an active position in v_{j-1} and thus a non-variable position in w_{j-1}. Note that w_{j-1} satisfies properties (1) and (2). Let $l'_j \to r'_j$ be the rule in $\overline{\mathcal{P} \cup \mathcal{R}}$ that is used to rewrite u_j to v_j. (So $l_j = \mathsf{unlabel}(l'_j)$ and $r'_j = \mathsf{label}(r_j)$.) Since w_{j-1} is linear, it follows that $w_{j-1}|_{p_j}$ unifies with l'_j. Let σ_j be an idempotent most general unifier of these two terms. Define $w_j = w_{j-1}[r'_j]_{p_j}\sigma_j$. Since w_{j-1} is linear, $\mathcal{V}\mathsf{ar}(w_{j-1}) \cap \mathcal{V}\mathsf{ar}(l'_j) = \varnothing$, and σ_j is idempotent, $(\mathcal{V}\mathsf{ar}(w_{j-1}) \setminus \mathcal{V}\mathsf{ar}(w_{j-1}|_{p_j})) \cap \mathcal{D}\mathsf{om}(\sigma_j) = \varnothing$ and thus $w_j = w_{j-1}[r'_j\sigma_j]_{p_j}$. Because w_{j-1} contains only active function symbols, $x\sigma_j =$

$\mathsf{label}(x\sigma_j)$ for all $x \in \mathcal{V}\mathsf{ar}(l'_j)$ and hence $w_j \in \mathcal{T}(\overline{\mathcal{F}}, \mathcal{V})$. Since $\mathcal{P}$ and $\mathcal{R}$ are right-linear, it follows that w_j is linear. We have $w_{j-1}[l'_j\sigma_j]_{p_j} \xrightarrow{\mathsf{a}} w_{j-1}[r'_j\sigma_j]_{p_j} = w_j$. It follows that w_j represents the active region of v_j and thus $v_j = w_j\tau_j$ for some substitution $\tau_j \colon \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$, which proves property (1). Property (2) holds by construction.

Since $v_n \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s'_i$, the active part of s'_i is the same as the active part of v_n. It follows that $s'_i = w_n\tau$ for some substitution $\tau \colon \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$. Since all rewrite steps in the infinite sequence starting from s'_i are strongly active, these steps can also be performed when starting from w_n. Removing all labels produces a minimal rewrite sequence starting at the term $w = \mathsf{unlabel}(w_n)$ with the desired properties. □

By using the statement of the previous proof we can easily show the correctness of Lemma 5.40.

Proof of Lemma 5.40. Assume that there is a minimal rewrite sequence

$$s_1 \xrightarrow{\epsilon}_{\alpha_1} t_1 \to^*_{\mathcal{R}} s_2 \xrightarrow{\epsilon}_{\alpha_2} t_2 \to^*_{\mathcal{R}} \cdots$$

in which infinitely many β-steps directly follow α-steps. According to Definition 5.39 we know that $\alpha_i = \alpha$ for infinitely many $i \geqslant 1$. Applying the statement of the previous proof with $s \to t = \alpha$ yields a term $w \in \mathsf{RFC}_{\mathsf{rhs}(\alpha)}(\mathcal{P} \cup \mathcal{R})$ such that

$$w \xrightarrow{\epsilon}_{\alpha_i} \cdot \to^*_{\mathcal{R}} \cdot \xrightarrow{\epsilon}_{\alpha_{i+1}} \cdot \to^*_{\mathcal{R}} \cdots$$

for some $i \geqslant 1$. Since the minimal rewrite sequence starting at s_i contains infinitely many situations in which β directly follows α, we know that this also holds for the minimal rewrite sequence starting at w. □

Index