Lazy Termination Analysis[1]

dissertation

by

Harald Zankl

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 "Doktor der technischen Wissenschaften"

advisor: Univ.-Prof. Dr. Aart Middeldorp

Innsbruck, September 2009

[1] This thesis is supported by FWF project P18763.

dissertation

Lazy Termination Analysis

Harald Zankl
Harald.Zankl@uibk.ac.at

September 2009

advisor: Univ.-Prof. Dr. Aart Middeldorp

To Mom and Dad

Abstract

This thesis is concerned with automated termination analysis of first-order term rewrite systems. "Lazy Termination Analysis" addresses various termination arguments and brings them to their full potential by encoding them as arithmetic constraints. Here lazy indicates that the actual work is done by somebody else, i.e., a SAT, PB, or SMT solver. Hence, unlike most contributions in this field, the thesis also goes beyond pure SAT solving by considering arithmetic constraints instead of plain propositional logic. How such constraints can be solved most efficiently by means of SAT, PB, or SMT techniques is also outlined. One brilliant example demonstrating the benefits of our approach is the Knuth-Bendix order, which—although already more than 40 years of age—still lacks an efficient implementation. It turned out that the method can quite succinctly be encoded as linear arithmetic constraints which can then most efficiently be solved by current SMT solvers. The expressiveness of SAT also allows to implement increasing interpretations, a variant of polynomial interpretations which at the same time also considers information from the dependency graph. As already suggested by its name this method allows some rules to temporarily increase the interpreted value. This thesis also provides theory and empirical evaluation for matrices over the reals which makes it the first contribution that considers reals encoded in SAT. That not only termination criteria are in the scope of arithmetic encodings is also demonstrated. To this end a looping reduction of given length involving only strings of given size is formulated as a satisfiability problem. Furthermore we formalized looping non-termination in the theorem prover Isabelle resulting in the first automated verifier capable of certifying non-termination. Finally after all the (non-)termination encodings we investigate the other direction, i.e., encode propositional satisfiability as a termination problem in rewriting. Only the most simple formulas yield rewrite systems that can be handled by sophisticated termination analyzers. Hence this approach allows to easily generate testbeds of challenging rewrite systems. All encodings presented in this thesis have been integrated into the fully automatic (non-)termination analyzer T$_{\text{T}}$T$_2$.

Acknowledgments

I am grateful to my advisor Aart Middeldorp for introducing me into research. Although he allowed me much freedom in my studies his valuable guidance and expertise was indispensable for the success and progress of this thesis. Also the other two T$_{\text{T}}$T$_2$ developers and office mates Martin Korp and Christian Sternagel deserve special thanks for their contributions and (programming) skills that have made T$_{\text{T}}$T$_2$ successful. Apart from them also Nao Hirokawa, Stefan Jörer, and Sarah Winkler contributed in some form or another to T$_{\text{T}}$T$_2$.

I must mention my two office mates here again, not only for the myriads of hours of helpful—and sometimes rather heavy—discussion concerning research but also for all the common activities outside the office. And Christian Sternagel even deserves a third mention here for being my main TEXnician; I would even consider calling him ChrisTEXan.

Although some of them have not directly contributed to this thesis I thank all members of the Computational Logic group at University of Innsbruck, namely Aart, Andreas, Christian, Friedrich, Georg, Martin A., Martin K., Martina, René, Sarah, and Simon for providing a group atmosphere which not necessarily ends after work as witnessed by various group activities such as group cinemas, group espressi, group lunches, group hikes, group holidays, group marathons, and group parties.

The financial support from the Austrian Fund of Science (FWF) over the last three years via the project P18763 is also acknowledged.

Finally, I thank my family and my companion in life Michaela for supporting my work on a topic which is completely incomprehensible to them.

Contents

I can resist everything, except termination.

freely adapted from Oscar Wilde (1854–1900)

Introduction

Term rewriting is a powerful model of computation which forms the underlying theory of declarative programming and theorem proving. Since term rewriting is Turing-complete, all basic properties such as termination and confluence are undecidable. Nevertheless researchers have spent much effort in developing (incomplete) methods to (dis-)prove termination of rewrite systems since this property is essential for, e.g., verification or Knuth-Bendix completion.

Starting more then 40 years ago, Knuth and Bendix introduced in a landmark paper the Knuth-Bendix order [48], one of the first termination criteria for rewrite systems. Since then for many years simplification orderings have dominated research although they are rather restrictive in power. It took until 1995 when Zantema introduced semantic labeling [96] as one method that overcomes the limitations of simplification orderings. Also the monotonic semantic path order [7] established by Borralleras *et al.* in 2000 yields enormous gains in power but is challenging to implement. At about the same time Arts and Giesl presented the dependency pair method [3] which is omnipresent nowadays.

Since the beginning of the 21^{st} century the focus turned on *automation* of termination criteria. In 2004 the first international competition[1] of termination tools emerged after a tool demonstration in 2003. Since then this event takes place annually and various tools compete against each other in different categories on standard testbenches. Nowadays the majority of the tools implement the dependency pair framework [31, 33, 79, 38] which is a modular extension of the dependency pair method [3]. An exploration of (the order of) recursive function calls is captured in the dependency graph which allows modular termination checking due to a recursive treatment of cyclic call structures [38] in this graph. Recently termination analysis of rewrite systems has successfully been applied in the termination analysis of functional [34] and logic programs [74, 68].

Kurihara and Kondo [57] were the first to encode a termination method for term rewriting into propositional logic. In this highly innovative paper they showed how to encode orientability with respect to the lexicographic path order (LPO) [45] as a satisfaction problem using binary decision diagrams [9]. In the recent past a vast number of SAT encodings has been proposed for various termination methods. Codish *et al.* [10] presented a more efficient formulation for the properties of a precedence. They achieve further speedup by replacing binary decision diagrams by state of the art SAT solvers. In [11, 91] encodings of argument filterings are presented which can be combined with propositional encodings of reduction pairs in order to obtain logic-based implementations of traditional processors in the dependency pair method. Encodings of other termination methods are described in [22–25, 41, 43, 50, 51, 75, 92, 99]. The

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

benefits of the SAT encodings seem almost endless. Existing termination criteria can usually be implemented with considerably less effort compared to dedicated algorithms. For many methods these encodings additionally result in a tremendous speedup [10, 11, 23, 75, 91]. And in addition SAT solving paves the way for exploring extremely large search spaces [41, 22, 92] which have previously been out of reach resulting in innovative and new termination criteria. This thesis is almost completely devoted to such encodings which improve termination tools in two aspects: power and speed. Since most computational effort is thus transferred from the termination analyzer to the constraint solver this explains the title of the thesis "Lazy Termination Analysis".

Overview

The remainder of the thesis is organized as follows.

Chapter 1 presents preliminaries concerning term rewriting in general and the dependency pair framework in particular. Furthermore a simple language for arithmetic constraints is fixed which is employed for all encodings. The chapter ends with a description of the test environment used for the experiments.

Chapter 2 is concerned with the Knuth-Bendix order. An easy encoding using arithmetic constraints is presented which allows to implement the order with considerably less effort compared to dedicated algorithms while yielding gains in efficiency. The encoding is augmented by argument filterings in the second part of the chapter which improves applicability of the order. Extensive experimental results and an assessment of the contribution conclude the chapter.

Chapter 3 introduces increasing interpretations which are a generalization of polynomial interpretations. The novelty of the approach is that some rules might increase the interpreted value which is in stark contrast to all existing termination criteria where at least a weak decrease is demanded for every rule considered. Two theorems are formulated for increasing interpretations and possibilities for an implementation are presented and empirically evaluated.

Chapter 4 revisits matrix interpretations. We show that matrices over the non-negative real numbers may be used for termination proofs and evaluate the approach in comparison with natural and non-negative rational numbers as coefficients.

Chapter 5 is concerned with non-termination. The first part of the chapter is devoted to finding loops. To this end an encoding is given that captures a loop of given length involving only strings of a given size. The second part of the chapter sketches an Isabelle [69] formalization of loops which has been integrated into the termination certifier IsaFoR/CeTA [80]. This contribution makes CeTA the first automated certifier capable of certifying non-termination of rewrite systems.

Chapter 6 is devoted to solving arithmetic constraints. They are not solved directly but transformed into problems over propositional satisfiability (SAT), pseudo-boolean logic (PB), or satisfiability modulo theory (SMT). For the SAT case it is shown how natural (integer, rational, and real) numbers are represented in binary and how necessary operations of arithmetic are mimicked.

Chapter 7 focuses on encoding SAT as a termination problem in rewriting. Since encoding termination criteria in SAT performs so well we explored the reverse direction. The outcome are three transformations from propositional formulas φ to rewrite systems $\mathcal{R}^\varphi$ such that $\mathcal{R}^\varphi$ is terminating if and only if φ is (un-)satisfiable. Surprisingly only rewrite systems stemming from extremely small formulas are in the reach of current termination tools. Hence this approach is very suitable for generating testbeds of rewrite systems where proving (non-)termination is very challenging.

Appendix A presents the termination analyzer $\mathsf{T_TT_2}$ with special focus on its strategy language.

Suggested Way of Reading

The chapters within this thesis can be read independently from each other with one exception: All chapters depend on the preliminaries presented in Chapter 1. Furthermore, Chapter 4 refers to some results from Section 3.1 but appropriate references are given there.

Chapter 1

Preliminaries

This chapter first recalls some basics of term rewriting in Section 1.1 (for more details we refer to [4, 100]) before introducing a simple version of the dependency pair framework in Section 1.2. The grammar of arithmetic constraints which are heavily used within this thesis to encode (non-)termination criteria is fixed in Section 1.3. All experiments presented within this document have been performed on the run time environment and testbeds described in Section 1.4.

Within this thesis $\mathbb{N}$ refers to the set $\{0, 1, \dots\}$ of natural numbers whereas $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ denote integer, rational, and real numbers, respectively. If confusion might arise, sometimes relations such as $<$ are indexed by the corresponding domain, like in $<_{\mathbb{Z}}$, the standard order on integers. Furthermore notation like $\mathbb{N}^{>3}$ defines the set $\{n \in \mathbb{N} \mid n > 3\}$.

1.1 Term Rewriting

A *signature* $\mathcal{F}$ is a set of function symbols with fixed arities. Let $\mathcal{V}$ denote an infinite set of *variables* disjoint from $\mathcal{F}$. Then $\mathcal{T}(\mathcal{F}, \mathcal{V})$ forms the set of *terms* over the signature $\mathcal{F}$ using variables from $\mathcal{V}$. For a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ the expression $\|t\|$ denotes the *number of function symbols* occurring in t, $|t|$ computes its *size*, $|t|_a$ for $a \in \mathcal{F} \cup \mathcal{V}$ counts how often the symbol a occurs in t, and $\mathcal{V}\mathsf{ar}(t)$ returns all *variables* occurring in t. A term $t = f(t_1, \dots, t_n)$ has *root* f. For a term t the set $\mathcal{P}\mathrm{os}(t)$ of *positions* in t is defined inductively by $\{\epsilon\}$ if $t \in \mathcal{V}$ and $\mathcal{P}\mathrm{os}(t) = \{\epsilon\} \cup \{ip \mid p \in \mathcal{P}\mathrm{os}(t_i), 1 \leqslant i \leqslant n\}$ if $t = f(t_1, \dots, t_n)$. For $p \in \mathcal{P}\mathrm{os}(t)$ the subterm of t at position p is denoted by $t|_p$. *Contexts* are terms over the extended signature $\mathcal{F} \cup \{\Box\}$ with exactly one occurrence of the fresh constant $\Box$ (called *hole*). The expression $C[t]$ denotes the result of replacing the hole in C by the term t. A *substitution* σ is a mapping from variables to terms and $t\sigma$ denotes the result of replacing the variables in t according to σ. As usual we assume that a substitution changes only finitely many variables which allows to write it as a finite set of bindings $\{x_1/t_1, \dots, x_n/t_n\}$. *Rewrite rules* are pairs of terms (l, r), usually written as $l \to r$, with the property that l is not a variable and that all variables of r appear in l. A *term rewrite system* (TRS) is a set of rewrite rules. We assume TRSs to be finite. A *string rewrite system* (SRS) is a TRSs where all function symbols are unary. A TRS $\mathcal{R}$ is *non-duplicating* if for all variables $x \in \mathcal{V}$ and rewrite rules $l \to r \in \mathcal{R}$ the condition $|l|_x \geqslant |r|_x$ holds. The *rewrite relation* induced by a TRS $\mathcal{R}$ is a binary relation on terms denoted by $\to_{\mathcal{R}}$ with $s \to_{\mathcal{R}} t$ for terms s and t if and only if there exist a rewrite rule $l \to r \in \mathcal{R}$, a context C, and a substitution σ

such that $s = C[l\sigma]$ and $t = C[r\sigma]$. The (reflexive and) transitive closure of $\to_{\mathcal{R}}$ is denoted by $(\to_{\mathcal{R}}^*)\ \to_{\mathcal{R}}^+$. A TRS $\mathcal{R}$ is called *strongly normalizing* (SN) or *terminating* if $\to_{\mathcal{R}}$ is well-founded, i.e., there is no rewrite sequence of the form $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} t_3 \to_{\mathcal{R}} t_4 \to_{\mathcal{R}} \cdots$. A rewrite sequence of the special shape $t_1 \to_{\mathcal{R}} \cdots \to_{\mathcal{R}} t_n \to_{\mathcal{R}} C[t_1\sigma]$ is called a *loop* of length n and can be represented by the triple $([t_1, \ldots, t_n], C, \sigma)$. Clearly every loop gives rise to an infinite rewrite sequence of the form $t_1 \to_{\mathcal{R}}^+ C[t_1\sigma] \to_{\mathcal{R}}^+ C[C[t_1\sigma]\sigma] \to_{\mathcal{R}}^+ \cdots$ since rewriting is closed under contexts and substitutions. In the sequel we omit the subscript $\mathcal{R}$ in $\to_{\mathcal{R}}$, $\to_{\mathcal{R}}^*$, and $\to_{\mathcal{R}}^+$ whenever $\mathcal{R}$ is irrelevant or clear from the context.

We illustrate some concepts by means of the following example.

Example 1.1. The TRS Cime/append_wrong[1] consisting of the seven rules

$$\begin{aligned}
\mathsf{is_empty}(\mathsf{nil}) &\to \mathsf{true} & \mathsf{is_empty}(\mathsf{cons}(x, xs)) &\to \mathsf{false}\\
\mathsf{tl}(\mathsf{cons}(x, xs)) &\to \mathsf{cons}(x, xs) & \mathsf{hd}(\mathsf{cons}(x, xs)) &\to x\\
\mathsf{if}(xs, ys, \mathsf{false}) &\to \mathsf{cons}(\mathsf{hd}(xs), \mathsf{tl}(xs) +\!\!+ ys) & \mathsf{if}(xs, ys, \mathsf{true}) &\to ys\\
xs +\!\!+ ys &\to \mathsf{if}(xs, ys, \mathsf{is_empty}(xs))
\end{aligned}$$

admits the non-terminating rewrite sequence

$$\begin{aligned}
&\mathsf{if}(\mathsf{cons}(x, xs), ys, \mathsf{false})\\
&\quad\to \mathsf{cons}(\mathsf{hd}(\mathsf{cons}(x, xs)), \mathsf{tl}(\mathsf{cons}(x, xs)) +\!\!+ ys)\\
&\quad\to \mathsf{cons}(\mathsf{hd}(\mathsf{cons}(x, xs)), \mathsf{cons}(x, xs) +\!\!+ ys)\\
&\quad\to \mathsf{cons}(\mathsf{hd}(\mathsf{cons}(x, xs)), \mathsf{if}(\mathsf{cons}(x, xs), ys, \mathsf{is_empty}(\mathsf{cons}(x, xs))))\\
&\quad\to \mathsf{cons}(\mathsf{hd}(\mathsf{cons}(x, xs)), \mathsf{if}(\mathsf{cons}(x, xs), ys, \mathsf{false}))\\
&\quad\to \cdots
\end{aligned}$$

forming a loop of length four with context $\mathsf{cons}(\mathsf{hd}(\mathsf{cons}(x, xs)), \Box)$ and empty substitution.

1.2 Dependency Pair Framework

In this section we sketch a simplified version of the dependency pair framework [3, 31, 33, 38, 79] which we specialize to fit our requirements. Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$. The signature $\mathcal{F}$ is extended with *dependency pair symbols* $f^\sharp$ for every symbol $f \in \{\mathrm{root}(l) \mid l \to r \in \mathcal{R}\}$, where $f^\sharp$ has the same arity as f, resulting in the signature $\mathcal{F}^\sharp$.[2] In examples one usually uses capitalization, i.e., one writes F for $f^\sharp$. 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 $l^\sharp \to t^\sharp$ is a *dependency pair* of $\mathcal{R}$. Here $l^\sharp$ and $t^\sharp$ are the result of replacing the root symbols in l and t by the corresponding dependency pair symbols. The set of dependency pairs of $\mathcal{R}$ is denoted by $\mathsf{DP}(\mathcal{R})$.

[1] Labels in sans-serif font refer to TRSs from the TPDB (cf. Section 1.4).
[2] Function symbols that appear as a root of a left-hand side are called *defined*.

Example 1.2. The TRS from Example 1.1 admits five dependency pairs:

$$xs +\!\!+^\sharp ys \to \mathsf{IS\_EMPTY}(xs) \tag{1.1}$$
$$xs +\!\!+^\sharp ys \to \mathsf{IF}(xs, ys, \mathsf{is\_empty}(xs)) \tag{1.2}$$
$$\mathsf{IF}(xs, ys, \mathsf{false}) \to \mathsf{TL}(xs) \tag{1.3}$$
$$\mathsf{IF}(xs, ys, \mathsf{false}) \to \mathsf{tl}(xs) +\!\!+^\sharp ys \tag{1.4}$$
$$\mathsf{IF}(xs, ys, \mathsf{false}) \to \mathsf{HD}(xs) \tag{1.5}$$

A *DP problem* $(\mathcal{P}, \mathcal{R})$ is a pair of TRSs $\mathcal{P}$ and $\mathcal{R}$ such that the root symbols of the rules in $\mathcal{P}$ do neither occur in $\mathcal{R}$ nor in proper subterms of the left- and right-hand sides of rules in $\mathcal{P}$. The problem is said to be *finite* if there is no infinite sequence $s_1 \to_\mathcal{P} t_1 \to^*_\mathcal{R} s_2 \to_\mathcal{P} t_2 \to^*_\mathcal{R} \cdots$ such that all terms t_1, t_2, ... are terminating with respect to $\mathcal{R}$. Such an infinite sequence is called *minimal*. The main result underlying the dependency pair approach states that termination of a TRS $\mathcal{R}$ is equivalent to finiteness of the *initial* DP problem $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$.

The concept of minimality is illustrated in the following example.

Example 1.3. Let $(\mathcal{P}, \mathcal{R})$ be the initial DP problem for the TRS from Example 1.1. Then

$$\begin{aligned} t = \mathsf{IF}(\mathsf{cons}(x, xs), ys, \mathsf{false}) \\ &\to_\mathcal{P} \mathsf{tl}(\mathsf{cons}(x, xs)) +\!\!+^\sharp ys \\ &\to^*_\mathcal{R} \mathsf{cons}(x, xs) +\!\!+^\sharp ys \\ &\to_\mathcal{P} \mathsf{IF}(\mathsf{cons}(x, xs), ys, \mathsf{is\_empty}(\mathsf{cons}(x, xs))) \\ &\to^*_\mathcal{R} \mathsf{IF}(\mathsf{cons}(x, xs), ys, \mathsf{false}) \\ &\to_\mathcal{P} \cdots \end{aligned}$$

is an infinite sequence. It is minimal since all reducts of t are terminating with respect to $\mathcal{R}$. For the substitution $\sigma = \{ys/\mathsf{if}(\mathsf{cons}(x, xs), ys, \mathsf{false})\}$ the term $t\sigma$ clearly starts an infinite sequence but no minimal one due to the fact that $\mathsf{if}(\mathsf{cons}(x, xs), ys, \mathsf{false})$ itself is not terminating (cf. Example 1.1).

In order to prove a DP problem finite, a number of *DP processors* have been developed. DP processors are functions that take a DP problem $(\mathcal{P}, \mathcal{R})$ as input and return a set of DP problems or "no" as output. In order to be employed to prove termination DP processors need to be *sound*, that is, if all DP problems in a set returned by a DP processor are finite then $(\mathcal{P}, \mathcal{R})$ 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 $(\mathcal{P}, \mathcal{R})$ is not finite. Proofs in the DP framework can be seen as trees. The nodes are DP problems and the children of a node $(\mathcal{P}, \mathcal{R})$ are the single DP problems (or "no") a DP processor returns when being applied to $(\mathcal{P}, \mathcal{R})$. The root node is the initial DP problem. Hence if only sound DP processors are used and all leaves in the proof tree have empty $\mathcal{P}$ components, then this shows termination of the original TRS. On the contrary, if along a

path within the proof tree only complete DP processors are applied and the leaf node is "no" then the original TRS is non-terminating.

One important DP processor is the dependency graph. In general it is not computable but sound approximations exist [3, 32, 38, 55, 65]. Here soundness means that every edge in the original graph also is an edge in the estimated graph and hence it forms an over-approximation of the actual dependency graph.

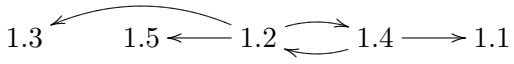
Definition 1.4. Let $(\mathcal{P}, \mathcal{R})$ be a DP problem. The *dependency graph* $\mathsf{DG}(\mathcal{P}, \mathcal{R})$ with nodes $\mathcal{P}$ contains an edge from node $s \to t$ to node $u \to v$ if there exist substitutions σ and τ such that $t\sigma \to_{\mathcal{R}}^* u\tau$.

Next we formulate a corresponding DP processor. Here a *strongly connected component* (SCC) is a maximal set of nodes such that there is a non-empty path from every node to every (not necessarily distinct) other node. Maximality means that the property of being an SCC is lost if a further node is added.

Theorem 1.5. *The following DP processor is sound and complete. For a DP problem $(\mathcal{P}, \mathcal{R})$ the processor returns $\{(\mathcal{P}_1, \mathcal{R}), \ldots, (\mathcal{P}_n, \mathcal{R})\}$ where $\mathcal{P}_1, \ldots, \mathcal{P}_n$ are the SCCs of $\mathsf{DG}(\mathcal{P}, \mathcal{R})$.* □

Next we compute the dependency graph for the running example.

Example 1.6. The initial DP problem from the TRS of Example 1.1 admits the dependency graph



with one SCC $\{1.2, 1.4\}$.

One popular method to get sound and complete DP processors are reduction pairs. We will encounter reduction pairs based on KBO and monotone algebras in Chapters 2 and 4, respectively. Formally, a *reduction pair* $(\gtrsim, >)$ consists of a rewrite pre-order $\gtrsim$ (a pre-order on terms that is closed under contexts and substitutions) and a well-founded order $>$ that is closed under substitutions such that the inclusion $> \cdot \gtrsim \subseteq >$ (compatibility) holds. Reduction pairs give rise to DP processors.

Theorem 1.7. *Let $(\gtrsim, >)$ be a reduction pair. The processor that maps a DP problem $(\mathcal{P}, \mathcal{R})$ to*

- $\{(\mathcal{P} \setminus >, \mathcal{R})\}$ *if* $\mathcal{P} \subseteq \gtrsim \cup >$ *and* $\mathcal{R} \subseteq \gtrsim$
- $\{(\mathcal{P}, \mathcal{R})\}$ *otherwise*

is sound and complete. □

For all reduction pairs originating from simplification orders the order $>$ is closed under contexts although the theorem above does not require this. To overcome this limitation we introduce argument filterings. An *argument filtering* for a signature $\mathcal{F}$ is a mapping π that assigns to every n-ary function

symbol $f \in \mathcal{F}$ an argument position $i \in \{1, \ldots, n\}$ or a (possibly empty) list $[i_1, \ldots, i_m]$ of argument positions with $1 \leqslant i_1 < \cdots < i_m \leqslant n$. The signature $\mathcal{F}^\pi$ consists of all function symbols f such that $\pi(f)$ is some list $[i_1, \ldots, i_m]$, where in $\mathcal{F}^\pi$ the arity of f is m. Every argument filtering π induces a mapping from $\mathcal{T}(\mathcal{F}, \mathcal{V})$ to $\mathcal{T}(\mathcal{F}^\pi, \mathcal{V})$, also denoted by π: $\pi(t) = t$ if $t \in \mathcal{V}$, $\pi(t) = \pi(t_i)$ if $t = f(t_1, \ldots, t_n)$ with $\pi(f) = i$, and $\pi(t) = f(\pi(t_{i_1}), \ldots, \pi(t_{i_m}))$ if $t = f(t_1, \ldots, t_n)$ with $\pi(f) = [i_1, \ldots, i_m]$. We further abuse notation and define $\pi(l \to r)$ for rewrite rules and $\pi(\mathcal{R})$ for TRSs in an obvious manner. We write $i \in \pi(f)$ if $\pi(f) = i$ or $\pi(f) = [i_1, \ldots, i_m]$ with $i_k = i$ for some $1 \leqslant k \leqslant m$.

The next example familiarizes the reader with argument filterings.

Example 1.8. Let π be an argument filtering satisfying $\pi({+\!\!+}^\sharp) = 2$ and $\pi(\mathsf{IF}) = [1, 2]$. Applying π to the SCC from the previous example results in $\pi(\{1.2, 1.4\}) = \{ys \to \mathsf{IF}(xs, ys), \mathsf{IF}(xs, ys) \to ys\}$. We have $2 \in \pi({+\!\!+}^\sharp)$ and $1, 2 \in \pi(\mathsf{IF})$ but $1 \notin \pi({+\!\!+}^\sharp)$.

Next we strengthen Theorem 1.7 since for a DP problem $(\mathcal{P}, \mathcal{R})$ a reduction pair does not necessarily have to weakly orient all rules from $\mathcal{R}$ due to minimality. It is safe to just consider the usable rules. The *usable rules modulo* π [33] for a DP problem $(\mathcal{P}, \mathcal{R})$ and an argument filtering π are denoted by $\mathcal{U}_\pi(\mathcal{P}, \mathcal{R})$ and defined as the union of all $\{l \to r \in \mathcal{R} \mid \mathrm{root}(l) \in \mathcal{US}_\pi(t)\}$ with $s \to t \in \mathcal{P}$. Here $\mathcal{US}_\pi(t)$ is defined as follows: If $t = x$ then $\mathcal{US}_\pi(t) = \varnothing$ and if $t = f(t_1, \ldots, t_n)$ then $\mathcal{US}_\pi(t)$ is the least set such that $f \in \mathcal{US}_\pi(t)$ and $\mathcal{US}_\pi(t_i) \subseteq \mathcal{US}_\pi(t)$ whenever $i \in \pi(f)$ for all $1 \leqslant i \leqslant n$. Furthermore, if $\mathrm{root}(l) \in \mathcal{US}_\pi(t)$ then $\mathcal{US}_\pi(r) \subseteq \mathcal{US}_\pi(t)$ for all $l \to r \in \mathcal{R}$.

Example 1.9. When computing the usable rules for the (filtered) SCC from the previous example the rules defining $\mathsf{is\_empty}$ are not usable since π ignores IF's third argument. Note that these rules are not usable independent from the choice of $\pi(\mathsf{is\_empty})$.

If reduction pairs are combined with usable rules, then *$\mathcal{C}_\mathcal{E}$-compatibility* must be ensured, i.e., for a fresh function symbol g the constraints $\mathsf{g}(x, y) \gtrsim x$ and $\mathsf{g}(x, y) \gtrsim y$ must hold (cf. [33, 39]). The following result allows to generalize Theorem 1.7 by usable rules and argument filterings.

Theorem 1.10. *Let $(\gtrsim, >)$ be a $\mathcal{C}_\mathcal{E}$-compatible reduction pair and π be an argument filtering. The processor that maps a DP problem $(\mathcal{P}, \mathcal{R})$ to*

- $\{(\mathcal{P} \setminus \{p \in \mathcal{P} \mid \pi(p) \in >\}, \mathcal{R})\}$ *if* $\pi(\mathcal{P}) \subseteq \gtrsim \cup >$ *and* $\pi(\mathcal{U}_\pi(\mathcal{P}, \mathcal{R})) \subseteq \gtrsim$
- $\{(\mathcal{P}, \mathcal{R})\}$ *otherwise*

is sound and complete. □

1.3 Encodings

This section describes a grammar for arithmetic constraints which allows to present the encodings from Chapters 2, 3, 4, and 5 concisely. Furthermore this section gives hints on design issues when translating arithmetic constraints to SAT. Information on solving such constraints can be found in Chapter 6.

Definition 1.11. An *arithmetic constraint* φ is described by the following BNF:

$$\begin{aligned}\varphi ::= \bot \mid \top \mid p \mid (\neg\varphi) \mid (\varphi \vee \varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \rightarrow \varphi) \mid (\varphi \leftrightarrow \varphi) \\ \mid (\alpha > \alpha) \mid (\alpha = \alpha) \mid (\alpha \geqslant \alpha)\end{aligned}$$

and

$$\alpha ::= \mathrm{a} \mid r \mid (\alpha + \alpha) \mid (\alpha - \alpha) \mid (\alpha \times \alpha) \mid (\varphi \ ?\ \alpha : \alpha)$$

where $\bot$ ($\top$) denotes propositional contradiction (tautology), p (a) ranges over propositional (arithmetical) variables, $\neg$ ($\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$) logical not (or, and, implication, bi-implication), $>$ ($=$, $\geqslant$) greater, (equal, greater or equal), r ranges over the real numbers, and $+$ ($-$, $\times$) denotes addition (subtraction, multiplication). If-then-else is written as $(\cdot\ ?\ \cdot\ :\ \cdot)$.

Next the reader is familiarized with the syntax of arithmetic constraints.

Example 1.12. The expressions

- 5
- p_{100}
- $(p_{10}\ ?\ (2.1 \times \mathrm{a}_{12}) : 0)$
- $((((\mathrm{a}_{12} + (\sqrt{2} \times \mathrm{a}_{30})) + 7.2) > (0 - \mathrm{a}_5)) \wedge p_2)$

are well-formed arithmetic constraints whereas

- $-\mathrm{a}_{10}$
- $(x \div 3)$
- $\mathrm{a} + 3$
- $((\mathrm{a} + 3)\wedge)$

are not.

To save parentheses we employ the binding hierarchy for the connectives where $\times$ binds strongest, followed by $+$ and $-$, which precede the relation symbols $>$, $\geqslant$, and $=$. The logical connective $\neg$ is next in the hierarchy, followed by $\vee$ and $\wedge$. The operators $\rightarrow$, $\leftrightarrow$, and $(\cdot\ ?\ \cdot\ :\ \cdot)$ bind weakest. Furthermore the operators $+$, $\times$, $\vee$, $\wedge$, and $\leftrightarrow$ are are left-associative while $-$ and $\rightarrow$ associate to the right. Using these conventions the most complex constraint from the example above simplifies to $\mathrm{a}_{12} + \sqrt{2} \times \mathrm{a}_{30} + 7.2 > 0 - \mathrm{a}_5 \ \wedge\ p_2$.

For solving arithmetic constraints we propose three different methods. The first approach considers a translation to satisfiability of propositional formulas (SAT) and works for any arithmetic constraint. In case of linear arithmetic there is the alternative to also use a satisfiability modulo theory solver (SMT, where the theory of choice is linear arithmetic) as back-end. Furthermore for

Chapter 2 we regard a third possibility namely pseudo-boolean satisfiability (PB) due to the fact that the Knuth-Bendix order can easily be represented in pseudo-boolean logic.

Considering the SAT back-end, any arithmetic constraint must be translated into propositional logic. To achieve this, for operations like + and × over the naturals we use similar encodings as [22] inspired by circuits used in hardware. For integers we favor two's complement representation due to the straightforward (re-)definitions of + and ×. This results in easier encodings than the approach in [23] where integers are dealt with in excess representation, i.e., every number is represented by $x - a$ where a is some fixed constant. For dealing with rationals our experiments produced consistent results with [25] where a fixed denominator (powers of two) is suggested. We also experimented with a fixed point representation of numbers which produced slightly worse results. To our knowledge there exists no SAT encoding for arithmetic over the reals. There are two main challenges which must be solved. First of all a representation of (a subset of) numbers from $\mathbb{R}$ must be fixed. Our proposal is to employ pairs (c, d) which represent $c + \sqrt{2}d$. Another major problem arises when two such numbers are compared, e.g., how to determine the value of $c + \sqrt{2}d > c' + \sqrt{2}d'$ if $c, c', d, d' \neq 0$? To this end heuristics are employed where an under- (over-approximation) of the left- (right-hand) side is used. More details on the translation from arithmetic to SAT and linear arithmetic to SMT are given in Chapter 6.

For the sake of completeness we want to mention the recent approach from [8] where SMT solvers are used to solve non-linear arithmetic. The idea is to fix a finite domain for every arithmetic variable. Afterwards the non-linear constraint is linearized by using an explicit case analysis of the values a variable might take. Adopting this approach one gets integer arithmetic for free (from the SMT solver) and [8] reports a speedup for rational valued domains. However, since SMT solvers currently do not support real numbers (just rationals) it is unclear how this contribution can be employed for arithmetic over the reals.

1.4 Test Environment and Testbenches

All encodings discussed within this thesis have been implemented on top of the Tyrolean Termination Tool 2 (T$_{\text{T}}$T$_2$) [56] version 1.01. MiniSat [20] (release from July 21, 2007) and MiniSat+ [19] (release from January 5, 2007) were used to check satisfiability of the SAT and PB based encodings and version 1.0.17 of Yices [18] was the choice for the SMT approach. All three tools are interfaced from T$_{\text{T}}$T$_2$ which is written in the functional programming language OCaml. For the SAT approach propositional formulas are transformed into conjunctive normal form (CNF) similar to [72] which is an improvement of the method presented in [82]. MiniSat expects the input formula to be in CNF. For more information about T$_{\text{T}}$T$_2$ (especially on its strategy language) please consider Appendix A. The loops within Chapter 5 have been certified with CeTA [80, 77] (version 1.04).

As a test set we used the 1391 TRSs and 732 SRSs of the standard rewriting

category in version 5.0 of the Termination Problems Data Base (TPDB), the testbed which is also used for the competitions. All tests have been performed on a server equipped with eight dual-core AMD Opteron® processors 885 running at a clock rate of 2.6 GHz and 64 GB of main memory. Computation time of the solvers was limited to 60 seconds per system. Apart from Chapter 5 $\mathsf{T}_\mathsf{T}\mathsf{T}_2$ was run on a single node only.

All details on the tests reported in this thesis are available from the URL

`http://colo6-c703.uibk.ac.at/ttt2/hz/thesis/`

Chapter 2

Knuth-Bendix Order

This chapter is concerned with proving termination of TRSs with the Knuth-Bendix order (KBO), a method invented by Knuth and Bendix in [48] well before termination research in term rewriting became a very popular and competitive endeavor. While checking if a given KBO orients a TRS can be performed in linear time [60], deciding KBO orientability is of polynomial effort [53]. The first termination tools that provide an implementation of KBO, AProVE [30] and TTT [39], adopt the algorithms in [17, 53] which are not easy to implement efficiently. Consequently AProVE and TTT never used KBO in the TRS category of the competitions while TTT2 relied on KBO in more than 20% of the successful termination proofs in the respective category of the 2008 competition. TTT2 implements the methods presented in the sequel. The aim of this chapter is to make KBO a more attractive choice for termination tools by presenting a simple[1] encoding of KBO orientability using arithmetic constraints such that checking satisfiability of the resulting constraints amounts to proving KBO orientation. Alternatively, an encoding in pseudo-boolean satisfiability (PB) is given. We show that in the case of KBO one can improve upon pure SAT encodings in two ways; on the one hand the implementation effort can be reduced by applying a more expressive constraint language, on the other hand performance can be improved by choosing the right back-end.

In Section 2.1 the necessary definitions for KBO are presented. Section 2.2 shows that weights can be bound from above. Then Section 2.3.1 introduces an encoding of KBO in the arithmetic constraint language introduced in Definition 1.11 and in pseudo-boolean logic (Section 2.3.2). The encoding is augmented by argument filterings in Section 2.4 before we compare the power and run times of our implementations with the ones of AProVE and TTT in Section 2.5 and show the enormous gain in efficiency. We draw some conclusions in Section 2.6.

The results of this chapter appeared in earlier publications: The SAT and PB encodings originate from [89] and the SAT encoding within the dependency pair setting was first published in [88]. These results have been augmented by a theorem on bounding weights from above and an SMT encoding in [91]. However, here we combine the SAT and SMT encodings by abstracting them into a single encoding using arithmetic constraints introduced in Definition 1.11.

[1] Here, simple should be understood in the sense of "easy to implement".

2.1 Preliminaries

A *quasi-precedence* $\gtrsim$ (*strict precedence* $>$) is a quasi-order (strict part of a quasi-order) on a signature $\mathcal{F}$. Sometimes we find it convenient to call a quasi-precedence simply precedence. A *weight function* for a signature $\mathcal{F}$ is a pair (w, w_0) consisting of a mapping $w\colon \mathcal{F} \to \mathbb{N}$ and a constant $w_0 > 0$ such that $w(c) \geqslant w_0$ for every constant $c \in \mathcal{F}$. Let $\mathcal{F}$ be a signature and (w, w_0) a weight function for $\mathcal{F}$. The *weight* of a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ is defined as follows:

$$w(t) = \begin{cases} w_0 & \text{if } t \text{ is a variable,} \\ w(f) + \sum_{i=1}^{n} w(t_i) & \text{if } t = f(t_1, \ldots, t_n). \end{cases}$$

A weight function (w, w_0) is *admissible* for a quasi-precedence $\gtrsim$ if $f \gtrsim g$ for all function symbols g whenever f is a unary function symbol with $w(f) = 0$.

Definition 2.1 ([48, 17, 76])**.** Let $\gtrsim$ be a quasi-precedence and (w, w_0) a weight function. We define the *Knuth-Bendix order* $>_{\mathsf{kbo}}$ on terms inductively as follows: $s >_{\mathsf{kbo}} t$ if $|s|_x \geqslant |t|_x$ for all variables $x \in \mathcal{V}$ and either

(a) $w(s) > w(t)$, or

(b) $w(s) = w(t)$ and one of the following alternatives holds:

(1) $t \in \mathcal{V}$, $s \in \mathcal{T}(\mathcal{F}^{(1)}, \{t\})$, and $s \neq t$, or

(2) $s = f(s_1, \ldots, s_n)$, $t = g(t_1, \ldots, t_m)$, $f \sim g$, and there exists an $1 \leqslant i \leqslant \min\{n, m\}$ with $s_i >_{\mathsf{kbo}} t_i$ and $s_j = t_j$ for all $1 \leqslant j < i$, or

(3) $s = f(s_1, \ldots, s_n)$, $t = g(t_1, \ldots, t_m)$, and $f > g$.

Here $\mathcal{F}^{(n)}$ denotes the set of all function symbols $f \in \mathcal{F}$ of arity n. Thus in case (b)(1) the term s consists of a non-empty sequence of unary function symbols applied to the variable t (since $s \neq t$ and $|s|_x \geqslant |t|_x$ for all $x \in \mathcal{V}$).

Specializing the above definition to (the reflexive closure of) a strict precedence, one obtains the definition of KBO in [4], except that we restrict weight functions to have range $\mathbb{N}$ instead of $\mathbb{R}^{\geqslant 0}$. According to results in [53, 59] this does not decrease the power of the order for finite TRSs.

Theorem 2.2. *A TRS $\mathcal{R}$ is terminating whenever there exist a quasi-precedence $\gtrsim$ and an admissible weight function (w, w_0) such that $\mathcal{R} \subseteq >_{\mathsf{kbo}}$.* □

Example 2.3. The TRS SK90/2.42 consisting of the rules

$$\begin{aligned}
\mathsf{flat}(\mathsf{nil}) &\to \mathsf{nil} & \qquad \mathsf{rev}(\mathsf{nil}) &\to \mathsf{nil} \\
\mathsf{flat}(\mathsf{unit}(x)) &\to \mathsf{flat}(x) & \qquad \mathsf{rev}(\mathsf{unit}(x)) &\to \mathsf{unit}(x) \\
\mathsf{flat}(x +\!\!+ y) &\to \mathsf{flat}(x) +\!\!+ \mathsf{flat}(y) & \qquad \mathsf{rev}(x +\!\!+ y) &\to \mathsf{rev}(y) +\!\!+ \mathsf{rev}(x) \\
\mathsf{flat}(\mathsf{unit}(x) +\!\!+ y) &\to \mathsf{flat}(x) +\!\!+ \mathsf{flat}(y) & \qquad \mathsf{rev}(\mathsf{rev}(x)) &\to x \\
\mathsf{flat}(\mathsf{flat}(x)) &\to \mathsf{flat}(x) & \qquad (x +\!\!+ y) +\!\!+ z &\to x +\!\!+ (y +\!\!+ z) \\
x +\!\!+ \mathsf{nil} &\to x & \qquad \mathsf{nil} +\!\!+ y &\to y
\end{aligned}$$

can be proved terminating by KBO. The weight function satisfying $w(\mathsf{flat}) = w(\mathsf{rev}) = w(+\!\!+) = 0$ and $w(\mathsf{unit}) = w(\mathsf{nil}) = w_0 = 1$ together with the quasi-precedence $\mathsf{flat} \sim \mathsf{rev} > \mathsf{unit} > +\!\!+ > \mathsf{nil}$ ensures that $l >_{\mathsf{kbo}} r$ for all rules $l \to r$. Using a quasi-precedence is essential to handle this system because the rules $\mathsf{flat}(x +\!\!+ y) \to \mathsf{flat}(x) +\!\!+ \mathsf{flat}(y)$ and $\mathsf{rev}(x +\!\!+ y) \to \mathsf{rev}(y) +\!\!+ \mathsf{rev}(x)$ demand $w(\mathsf{flat}) = w(\mathsf{rev}) = 0$ but KBO with strict precedence does not allow different unary functions to have weight zero.

One can imagine a more general definition of KBO. For instance, in case (b)(2) we could demand that $s_j \sim_{\mathsf{kbo}} t_j$ for all $1 \leqslant j < i$ where $s \sim_{\mathsf{kbo}} t$ denotes syntactic equality with respect to equivalent function symbols of the same arity. Here, function symbols f and g are equivalent if both, $f \sim g$ and $w(f) = w(g)$. Another obvious extension would be to compare the arguments according to permutations [76, 73, 75] or as multisets [87, 75]. To keep the discussion and implementation simple, we do not consider such refinements in the sequel. Furthermore, we stress that the main ingredient of KBO is the weight function which is rather unaffected by a status.

2.2 A Bound on Weights

We give a bound on weights to finitely characterize KBO orientability. While there are at most finitely many precedences on a finite signature, the following example demonstrates that there exist TRSs which need arbitrarily large weights.

Example 2.4. Consider the parametrized TRS consisting of the three rules

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

where $\mathsf{h}^0(x) = x$ and $\mathsf{h}^{n+1}(x) = \mathsf{h}(\mathsf{h}^n(x))$. Since the first rule duplicates the function symbol f we must assign weight zero to it. The admissibility condition for the weight function demands that f is a maximal element in the precedence. The second rule ensures that the weight of h is strictly larger than zero. It follows that the minimum weight of $\mathsf{h}^k(x)$ is $k + w_0$, which at the same time is the minimum weight of $\mathsf{i}(x)$. Thus $w(\mathsf{i})$ is at least k.

Throughout this section we do not distinguish vectors from matrices. We write $\mathbf{e}_i$ for the unit column vector whose i-th position is 1 and all other positions are 0 (the length of the vector is usually clear from the context). Let $A = (a_{ij})_{ij}$ be an $m \times n$ matrix. We define $\|A\| = \max_{i,j} |a_{ij}|$. The i-th row vector of A is denoted by $\mathbf{a}_i$. We say that a vector $\mathbf{x}$ is a solution of A if $A\mathbf{x} \geqslant \mathbf{0}$ and $\mathbf{x} \geqslant \mathbf{0}$. A solution $\mathbf{x}$ that maximizes $\{i \mid \mathbf{a}_i\mathbf{x} > \mathbf{0}\}$ with respect to set inclusion is called *principal*. Unless stated otherwise, matrix entries are integers.

Lemma 2.5. *Let A be an $m \times n$ matrix. There exists a principal solution $\mathbf{x}$ of A with $\|\mathbf{x}\| \leqslant n^{2^m}(2n\|A\|)^{2^m-1}$.*

Before proving the lemma we recall the idea for solving KBO from [17] and mention its consequences.

Example 2.6. Below on the left we give the inequations that the algorithm in [17] starts with (corresponding to a KBO proof attempt with empty precedence) for the TRS from Example 2.4 where we fix the parameter $k = 2$. The first four equations ensure that every weight is non-negative, the fifth equation captures w_0 and the last three equations express that for every rule $l \to r$ we have $w(l) > w(r)$:

$$\begin{array}{rcl} w(\mathsf{f}) & \geqslant & 0 \\ w(\mathsf{g}) & \geqslant & 0 \\ w(\mathsf{h}) & \geqslant & 0 \\ w(\mathsf{i}) & \geqslant & 0 \\ w_0 & > & 0 \\ w(\mathsf{f}) + w(\mathsf{g}) + 2w_0 & > & w(\mathsf{g}) + 2w(\mathsf{f}) + 2w_0 \\ w(\mathsf{h}) + w_0 & > & 2w(\mathsf{f}) + w_0 \\ w(\mathsf{i}) + w_0 & > & 2w(\mathsf{h}) + w_0 \end{array} \qquad \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 \\ -2 & 0 & 1 & 0 & 0 \\ 0 & 0 & -2 & 1 & 0 \end{pmatrix} \begin{pmatrix} w(\mathsf{f}) \\ w(\mathsf{g}) \\ w(\mathsf{h}) \\ w(\mathsf{i}) \\ w_0 \end{pmatrix} \geqslant \mathbf{0}$$

To solve the inequations on the left, the algorithm in [17] starts with the slightly generalized equational system $A\mathbf{x} \geqslant \mathbf{0}$ on the right, which clearly has a (principal) solution. But for a principal solution the algorithm must test if every strict inequation $w(s) > w(t)$ for corresponding terms s and t is indeed satisfied. If this is not the case then it replaces the inequation $w(s) > w(t)$ by $w(s) \geqslant w(t)$, and

(a) fails, if $s \in \mathcal{V}$, $t \in \mathcal{V}$, or $s = t$,

(b) adds the inequation $w(s_i) > w(t_i)$ for the least i ($1 \leqslant i \leqslant n$) such that $s_i \neq t_i$ if $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$, or

(c) extends the strict precedence ([17] only supports strict precedences) if possible by
   - $\mathrm{root}(s) > f$ for all $f \in \mathcal{F} \setminus \{\mathrm{root}(s)\}$ if $\mathrm{root}(s)$ is unary and all principal solutions require $w(\mathrm{root}(s)) = 0$, or
   - $\mathrm{root}(s) > \mathrm{root}(t)$ otherwise

and again tries to solve the inequations. In the example no principal solution satisfies the constraint $w(\mathsf{f}(\mathsf{g}(x, y))) > w(\mathsf{g}(\mathsf{f}(x), \mathsf{f}(y)))$ (since it simplifies to $0 > w(\mathsf{f})$ contradicting $w(\mathsf{f}) \geqslant 0$) but case (c) applies and the corresponding rule is oriented by extending the precedence with $\mathsf{f} > \mathsf{g}, \mathsf{h}, \mathsf{i}$. Since now there exists a principal solution satisfying the current constraints (e.g. $w(\mathsf{f}) = w(\mathsf{g}) = 0$, $w(\mathsf{h}) = w_0 = 1$, and $w(\mathsf{i}) = 3$) the algorithm successfully terminates.

The question remains which matrix A to take for Lemma 2.5. The example above demonstrates that A changes during the execution of the algorithm. Unfortunately not even the dimension of A stays constant; the columns of A are fixed by the number of unknowns but the rows may increase (cf. case (b) above). It is easy to see that the largest dimension of a matrix can be $m \times n$ where $n = |\mathcal{F}| + 1$ and $m = n + \sum_{l \to r \in \mathcal{R}} \min\{\mathrm{depth}(l), \mathrm{depth}(r)\}$ where $\mathrm{depth}(x) = 1$ if $x \in \mathcal{V}$ and $\mathrm{depth}(f(t_1, \ldots, t_q)) = 1 + \max\{\mathrm{depth}(t_i) \mid 1 \leqslant i \leqslant q\}$. Furthermore for every matrix A that might evolve during the algorithm we

have $\|A\| \leqslant \max\{|l|_a, |r|_a \mid l \to r \in \mathcal{R}, a \in \mathcal{F} \cup \mathcal{V}\}$. According to the lemma one can find a principal solution in $[0, n^{2^m}(2n\|A\|)^{2^m-1}]^n$. Hence we get $n^{2^m}(2n\|A\|)^{2^m-1}$ as an upper bound on the weights. Later this number will be referred to as $B_{\mathcal{R}}$. For Example 2.6 we get $n = 5$, $m = 5 + 7 = 12$, $\|A\| \leqslant 2$, and consequently $B_{\mathcal{R}} = 5^{2^{12}}20^{2^{12}-1}$. Section 2.5 shows that in practice much smaller weights suffice. Expressed in terms of the size of the TRS $\mathcal{R}$, the inequality $B_{\mathcal{R}} \leqslant N^{4^{N+1}}$ can easily be shown for $N = 1 + \sum_{l\to r\in\mathcal{R}}(|l| + |r|)$, provided that all symbols from $\mathcal{F}$ appear in $\mathcal{R}$.

In order to prove Lemma 2.5 we first recall the method of complete description (MCD) introduced by Dick *et al.* in [17].

Definition 2.7. For a row vector $(a_1, \ldots, a_n)$ we define the matrix $(a_1, \ldots, a_n)^\kappa$ as

$$(\mathbf{e}_i \mid a_i \geqslant 0) +\!\!+ (a_j\mathbf{e}_i - a_i\mathbf{e}_j \mid a_i < 0, a_j > 0)$$

with unit vectors $\mathbf{e}_i$, $\mathbf{e}_j$ of length n. The operator $+\!\!+$ merges vectors into a matrix. Let A be an $m \times n$ matrix. For each $0 \leqslant i \leqslant m$ we inductively define S_i^A as follows: S_0^A is the $n \times n$ identity matrix and $S_{i+1}^A = S_i^A(\mathbf{a}_{i+1}S_i^A)^\kappa$. The sum of all column vectors of S_m^A is denoted by $\mathbf{s}^A$.

Proposition 2.8 ([17])**.** *Let $A\mathbf{x} \geqslant \mathbf{0}$. Then $\mathbf{s}^A$ is a principal solution of A.* □

The next example demonstrates how to compute $(\cdot)^\kappa$ and $\mathbf{s}^A$.

Example 2.9. For the matrix $A = \begin{pmatrix}-2 & 0 & 1 & 3 & -1\end{pmatrix}$ we have

$$S_1^A = S_0^A(\mathbf{a}_1S_0^A)^\kappa = \mathbf{a}_1^\kappa = \begin{pmatrix}0&0&0&1&3&0&0\\1&0&0&0&0&0&0\\0&1&0&2&0&1&0\\0&0&1&0&2&0&1\\0&0&0&0&0&1&3\end{pmatrix} \text{ and } \mathbf{s}^A = \begin{pmatrix}4\\1\\4\\4\\4\end{pmatrix}.$$

We show that $\mathbf{s}^A$ fulfills the condition of Lemma 2.5.

Lemma 2.10. *Let B be a $p \times q$ matrix, C a $q \times r$ matrix and $\mathbf{a}$ a $1 \times q$ row vector. Then the following holds:*

1. $\|BC\| \leqslant q\|B\|\|C\|$,
2. $\|B\mathbf{a}^\kappa\| \leqslant 2\|B\|\|\mathbf{a}^\kappa\|$,
3. $\|\mathbf{a}^\kappa\| = \|\mathbf{a}\|$,
4. *the number of rows of $\mathbf{a}^\kappa$ is q and the number of columns is bounded from above by q^2.*

Proof. Claims 1 and 3 are trivial. Claim 2 follows from the fact that all columns in $\mathbf{a}^\kappa$ have at most two non-zero entries. For Claim 4 we reason as follows. The vector $\mathbf{a}$ is partitioned into three sets $\{a_i > 0\}$, $\{a_i = 0\}$, and $\{a_i < 0\}$ with cardinalities c, d, and e. We have $q = c + d + e$ and by construction $\mathbf{a}^\kappa$ has $c + d + ce$ columns. Let $\div$ denote integer division. One easily verifies that $c = (q + 1) \div 2$, $d = 0$, and $e = q \div 2$ maximizes this number. Since $((q + 1) \div 2)((q + 2) \div 2) \leqslant q^2$, this gives the desired result. □

Lemma 2.11. *If A is an $m \times n$ matrix then S_i^A is an $n \times q$ matrix for some $q \leqslant n^{2^i}$.*

Proof. We perform induction on i. The base case is trivial since S_0^A is the $n \times n$ identity matrix. Here $q = n \leqslant n^{2^0} = n$. For the inductive step assume that S_i^A is an $n \times q$ matrix with $q \leqslant n^{2^i}$. We show that S_{i+1}^A is an $n \times q'$ matrix for some $q' \leqslant n^{2^{i+1}}$. By definition $S_{i+1}^A = S_i^A(\mathbf{a}_{i+1}S_i^A)^\kappa$. Since $\mathbf{a}_{i+1}$ is a $1 \times n$ matrix and by assumption S_i^A is an $n \times q$ matrix, $\mathbf{a}_{i+1}S_i^A$ is of dimension $1 \times q$. By Lemma 2.10 point 4 we get that $(\mathbf{a}_{i+1}S_i^A)^\kappa$ is a $q \times q'$ matrix with $q' \leqslant q^2$. Hence $q' \leqslant q^2 \leqslant (n^{2^i})^2 = n^{2^{i+1}}$ by the induction hypothesis and the result immediately follows. □

Lemma 2.12. $\|S_i^A\| \leqslant (2n\|A\|)^{2^i-1}$.

Proof. We perform induction on i. The base case is trivial, because S_0^A is the identity matrix and thus $\|S_0^A\| = 1 = (2n\|A\|)^{2^0-1}$. We show the inductive step. Since $S_{i+1}^A = S_i^A(\mathbf{a}_{i+1}S_i^A)^\kappa$, we get

$$\begin{aligned}
\|S_{i+1}^A\| &\leqslant 2\|S_i^A\|\|(\mathbf{a}_{i+1}S_i^A)^\kappa\| \\
&= 2\|S_i^A\|\|\mathbf{a}_{i+1}S_i^A\| \\
&\leqslant 2n\|\mathbf{a}_{i+1}\|\|S_i^A\|^2 \\
&\leqslant 2n\|A\|\|S_i^A\|^2 \\
&\leqslant 2n\|A\|((2n\|A\|)^{2^i-1})^2 \\
&= (2n\|A\|)^{2^{i+1}-1}.
\end{aligned}$$

Here we used the (in-)equalities from Lemma 2.10, the trivial observation $\|\mathbf{a}_{i+1}\| \leqslant \|A\|$ in the fourth step, and the induction hypothesis in the fifth step. □

Now we are ready to prove the main lemma of this section.

Proof of Lemma 2.5. Easy consequence of Proposition 2.8 and Lemmata 2.11 and 2.12. □

Considering that $\mathbf{s}^A$ is an integer vector whenever A is an integer matrix, we obtain a finite characterization of KBO orientability.

Theorem 2.13. *Termination of $\mathcal{R}$ can be shown by KBO if and only if termination of $\mathcal{R}$ can be shown by KBO whose weights belong to $\{0, 1, \ldots, B_\mathcal{R}\}$.* □

We conclude this section by showing that a principal solution of A can be computed in polynomial time and mention its consequences.

Lemma 2.14. *Let $\mathbf{s}_i$ $(1 \leqslant i \leqslant m)$ be a solution of $A\mathbf{x} \geqslant \mathbf{e}_i$ if such a solution exists and $\mathbf{s}_i = \mathbf{0}$ otherwise. Then $\mathbf{s}_1 + \cdots + \mathbf{s}_m$ is a principal solution of A.*

Proof. Clearly $A\mathbf{s}_i \geqslant \mathbf{0}$ for $1 \leqslant i \leqslant m$. By Gale's theorem [26, Theorem 2.13] also $\mathbf{s}_1 + \cdots + \mathbf{s}_m$ is a solution of A. It is principal by construction. □

Therefore finding a principal solution of A boils down to solving $A\mathbf{x} \geqslant \mathbf{e}_i$ for $1 \leqslant i \leqslant m$. The latter can be handled in polynomial time due to the following known result.

Proposition 2.15 ([47, 46])**.** *$A\mathbf{x} \geqslant \mathbf{b}$ can be solved in polynomial time.* □

Note that a (possibly rational) solution $\mathbf{s}$ satisfying $A\mathbf{s} \geqslant \mathbf{e}_i$ can be transformed into the desired integer solution by multiplication with a sufficiently large scalar since $\mathbf{e}_i \geqslant \mathbf{0}$. Hence the approach in [17] can solve KBO in polynomial time (if MCD is replaced by linear programming) due to a similar argumentation as in [53]: The algorithm performs polynomially many steps, all matrices A that might appear during the algorithm are of polynomial size (cf. the discussion before Definition 2.7), a principal solution of A can be computed in polynomial time, and testing a finite precedence for well-foundedness (by computing its transitive closure and testing for irreflexivity) is polynomial.

2.3 Direct Encodings

In this section the constraints KBO puts on a TRS are transformed into (linear) arithmetic (Section 2.3.1) and pseudo-boolean logic (Section 2.3.2).

2.3.1 KBO in (Linear) Arithmetic

Due to the rich input format for arithmetic constraints (cf. Definition 1.11) one can directly translate orientability of KBO into constraints of (linear) arithmetic. As shown in Chapter 6 the encoding can be solved with both, SAT and SMT back-ends.

We employ for every function symbol $f \in \mathcal{F}$ non-negative integer variables w_f and p_f indicating the weight of f and its position in the precedence, respectively. Consequently an assignment satisfying $\mathrm{p}_f > \mathrm{p}_g$ indicates $f > g$.[2] Together with an integer variable w_0 the definition of an admissible weight function $\mathsf{ADM}(\mathcal{F})$ is easy as can be seen in the next definition.

Definition 2.16. For a signature $\mathcal{F}$, let $\mathsf{ADM}(\mathcal{F})$ be the constraint

$$\mathrm{w}_0 > 0 \;\wedge \bigwedge_{c \in \mathcal{F}^{(0)}} \mathrm{w}_c \geqslant \mathrm{w}_0 \;\wedge \bigwedge_{f \in \mathcal{F}^{(1)}} \big(\mathrm{w}_f = 0 \;\rightarrow \bigwedge_{g \in \mathcal{F}} (\mathrm{p}_f \geqslant \mathrm{p}_g)\big).$$

Similarly, the weight of a term can be computed.

Definition 2.17. The weight of a term t is encoded as follows:

$$W_t \;=\; \begin{cases} \mathrm{w}_0 & \text{if } t \in \mathcal{V}, \\ \mathrm{w}_f + \sum_{i=1}^{n} W_{t_i} & \text{if } t = f(t_1, \ldots, t_n). \end{cases}$$

We are now ready to define an arithmetic constraint that reflects the definition of $>_{\mathsf{kbo}}$.

[2] Codish *et al.* [10] were the first who encoded a precedence by a mapping into $\mathbb{N}$.

Definition 2.18. Let s and t be terms. We define the formula $\mathsf{KBO}(s >_{\mathsf{kbo}} t)$ as follows. If $s \in \mathcal{V}$ or $s = t$ or $|s|_x < |t|_x$ for some $x \in \mathcal{V}$ then $\mathsf{KBO}(s >_{\mathsf{kbo}} t) = \bot$. Otherwise

$$\mathsf{KBO}(s >_{\mathsf{kbo}} t) \;=\; W_s > W_t \;\vee\; \big(W_s = W_t \;\wedge\; \mathsf{KBO}(s >_{\mathsf{kbo}'} t)\big)$$

with $\mathsf{KBO}(s >_{\mathsf{kbo}'} t) = \top$ if $t \in \mathcal{V}$, $s \in \mathcal{T}(\mathcal{F}^{(1)}, \{t\})$, and $s \neq t$, and

$$\mathsf{KBO}(s >_{\mathsf{kbo}'} t) \;=\; \mathrm{p}_f > \mathrm{p}_g \vee \big(\mathrm{p}_f = \mathrm{p}_g \wedge \mathsf{KBO}(s_i >_{\mathsf{kbo}} t_i)\big)$$

in case of $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ where i denotes the least $1 \leqslant j \leqslant \min\{n, m\}$ with $s_j \neq t_j$.

Definition 2.19. Let $\mathcal{R}$ be a TRS. The formula $\mathsf{KBO}(\mathcal{R})$ is defined as

$$\mathsf{ADM}(\mathcal{F}) \;\wedge \bigwedge_{l \to r \in \mathcal{R}} \mathsf{KBO}(l >_{\mathsf{kbo}} r).$$

Theorem 2.20. *Termination of $\mathcal{R}$ can be shown by KBO if and only if the arithmetic constraint $\mathsf{KBO}(\mathcal{R})$ is satisfiable.* □

Note that for solving the above constraint two possibilities are proposed in Chapter 6. As discussed there, for the SAT approach numbers must be represented in binary. The variables p_f indicate the position of f in the precedence and hence can be bound by $\lceil \log_2(|\mathcal{F}|) \rceil$. For the variables of weights (w_f) such a bound is also possible due to Theorem 2.13. Section 2.5 shows that in practice much smaller weights suffice. The SMT approach is not affected by such considerations.

2.3.2 KBO in Pseudo-Boolean

A *pseudo-boolean constraint* (PBC) is of the form

$$\big(\sum_{i=1}^{n} a_i * x_i\big) \circ m$$

where $a_1, \ldots, a_n, m$ are fixed integers, $x_1, \ldots, x_n$ Boolean variables that range over $\{0, 1\}$, and $\circ \in \{\geqslant, =, \leqslant\}$. We separate PBCs that are written on a single line by semicolons. A sequence of PBCs is satisfiable if there exists an assignment which satisfies every PBC in the sequence. This means that PB can easily encode conjunctions of linear arithmetic expressions whereas disjunctions are tricky. In the sequel we show that nevertheless PBCs allow to encode KBO concisely. Since 2005 pseudo-boolean evaluation[3] is a track of the international SAT competition.[4]

In order to give an encoding of KBO orientability, we must take care of representing a precedence and a weight function. For the former we introduce three sets of propositional variables $X = \{X_{fg} \mid f, g \in \mathcal{F}\}$, $Y = \{Y_{fg} \mid f, g \in \mathcal{F}\}$,

[3] `http://www.cril.univ-artois.fr/PB07/`
[4] `http://www.satcompetition.org/`

and $Z = \{Z_{fg} \mid f, g \in \mathcal{F}\}$ depending on the underlying signature $\mathcal{F}$ [57, 10]. The intended semantics of these variables is that an assignment which satisfies a variable X_{fg} corresponds to a precedence with $f > g$, similarly Y_{fg} suggests $f \sim g$ and Z_{fg} takes the third possibility (either $g > f$ or f and g are incomparable). When dealing with strict precedences it is safe to assign zero to all Y_{fg} variables. We remark that the Z_{fg} variables are not necessary as far as termination proving power is concerned (because total precedences suffice as KBO is incremental with respect to the precedence) but they are essential to encode partial precedences which are sometimes handy (cf. Section 2.6).

Definition 2.21. For a signature $\mathcal{F}$ we define $\mathsf{PREC\text{-}PB}(\mathcal{F})$ as the conjunction of the PBCs below, for all $f, g \in \mathcal{F}$:

$$\begin{aligned} 2 * X_{fg} + Y_{fg} + Y_{gf} + 2 * Z_{fg} &= 2 \\ -X_{fg} + 2^l * Y_{fg} + 2^l * Z_{fg} + \overline{p}_l(f) - \overline{p}_l(g) &\geqslant 0 \\ 2^l * X_{fg} + Y_{fg} + 2^l * Z_{fg} + \overline{p}_l(f) - \overline{p}_l(g) &\geqslant 1 \end{aligned}$$

where $l = \lceil \log_2(|\mathcal{F}|) \rceil$ and $\overline{p}_l(f) = 2^{l-1} * f_l + \cdots + 2^0 * f_1$ denotes the position of f in the precedence by interpreting f in $\mathbb{N}$ using l bits. Here $f_l, \ldots, f_1$ are Boolean variables.

The above definition expresses all requirements of a quasi-precedence. The symmetry of $\sim$ and the mutual exclusion of the X, Y, and Z variables is mimicked by the first constraint. The second constraint encodes the conditions that are put on the X variables. Whenever a system needs $f > g$ in the precedence to be terminating then X_{fg} must evaluate to one and (because they are mutually exclusive) Y_{fg} and Z_{fg} to zero. Hence in order to remain satisfiable $\overline{p}_l(f) > \overline{p}_l(g)$ must hold. In a case where $f > g$ is not needed (but the TRS is KBO orientable) the constraint must remain satisfiable. Thus Y_{fg} or Z_{fg} evaluate to one and because $\overline{p}_l(g)$ is bound by $2^l - 1$ the constraint does no harm. Summing up, the second constraint encodes a proper order on the symbols in $\mathcal{F}$. The third constraint forms an equivalence relation on $\mathcal{F}$ using the Y_{fg} variables. Whenever $f \sim g$ is demanded somewhere in the encoding, then X_{fg} and Z_{fg} evaluate to zero by the first constraint. Satisfiability of the third constraint implies $\overline{p}_l(f) \geqslant \overline{p}_l(g)$ but at the same time symmetry demands that Y_{gf} also evaluates to one which leads to $\overline{p}_l(g) \geqslant \overline{p}_l(f)$ and thus to $\overline{p}_l(f) = \overline{p}_l(g)$.

The next definition captures the admissibility condition of a weight function for a signature $\mathcal{F}$.

Definition 2.22. For a signature $\mathcal{F}$ we denote by $\mathsf{ADM\text{-}PB}_k(\mathcal{F})$ the collection of PBCs

- $\overline{w_0}_k \geqslant 1$
- $\overline{w}_k(c) - \overline{w_0}_k \geqslant 0$ for all $c \in \mathcal{F}^{(0)}$
- $n * \overline{w}_k(f) + \sum_{f,g \in \mathcal{F}}(X_{fg} + Y_{fg}) \geqslant n$ for all $f \in \mathcal{F}^{(1)}$

where $n = |\mathcal{F}|$, $\overline{w}_k(f) = 2^{k-1} * f'_k + \cdots + 2^0 * f'_1$ denotes the weight of f in $\mathbb{N}$ using k bits, and $\overline{w_0}_k$ captures w_0. Again, $f'_k, \ldots, f'_1$ are Boolean variables.

In the definition above the first two PBCs express that w_0 is strictly larger than zero and that every constant has weight at least w_0. Whenever the considered function symbol f has weight larger than zero the third constraint is trivially satisfied. In the case that the unary function symbol f has weight zero the constraints on the precedence add up to n if and only if f is a maximal element. Note that X_{fg} and Y_{fg} are mutual exclusive (which is ensured when encoding the constraints on a quasi-precedence, cf. Definition 2.21).

For the encoding of $\mathsf{KBO\text{-}PB}_k(s >_{\mathsf{kbo}} t)$ and $\mathsf{KBO\text{-}PB}_k(s >_{\mathsf{kbo}'} t)$ (case (b) in Definition 2.1) auxiliary propositional variables $\mathsf{KBO}(s,t)$ and $\mathsf{KBO}'(s,t)$ are introduced. The intended meaning is that if $\mathsf{KBO}(s,t)$ ($\mathsf{KBO}'(s,t)$) evaluates to one under a satisfying assignment then $s >_{\mathsf{kbo}} t$ ($s >_{\mathsf{kbo}'} t$). The general idea of the encoding is very similar to the one from the previous subsection. As we do not know anything about the weights and the precedence at the time of encoding we have to consider the cases $w(s) > w(t)$ and $w(s) = w(t)$ at the same time. That is why $\mathsf{KBO}'(s,t)$ and the recursive call to $\mathsf{KBO\text{-}PB}_k(s >_{\mathsf{kbo}'} t)$ must be considered in any case.

The weight $\overline{w}_k(t)$ of a term t is defined similarly as in Definition 2.17 with the only difference that the weight $w(f)$ of the function symbol $f \in \mathcal{F}$ is represented in k bits as described in Definition 2.22.

Definition 2.23. Let s and t be terms. The encoding of $\mathsf{KBO\text{-}PB}_k(s >_{\mathsf{kbo}} t)$ amounts to $\mathsf{KBO}(s,t) = 0$ if $s \in \mathcal{V}$ or $s = t$ or $|s|_x < |t|_x$ for some $x \in \mathcal{V}$. In all other cases $\mathsf{KBO\text{-}PB}_k(s >_{\mathsf{kbo}} t)$ is the combination of $\mathsf{KBO\text{-}PB}_k(s >_{\mathsf{kbo}'} t)$ and

$$-(p+1) * \mathsf{KBO}(s,t) + \overline{w}_k(s) - \overline{w}_k(t) + \mathsf{KBO}'(s,t) \geqslant -p \tag{2.1}$$

where $p = 2^k * |t|$. Here $\mathsf{KBO\text{-}PB}_k(s >_{\mathsf{kbo}'} t)$ is the empty constraint when $t \in \mathcal{V}$, $s \in \mathcal{T}(\mathcal{F}^{(1)}, \{t\})$, and $s \neq t$. In the remaining case when $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ then $\mathsf{KBO\text{-}PB}_k(s >_{\mathsf{kbo}'} t)$ is the combination of the PBCs $\mathsf{KBO\text{-}PB}_k(s_i >_{\mathsf{kbo}} t_i)$ and

$$-2 * \mathsf{KBO}'(s,t) + 2 * X_{fg} + Y_{fg} + \mathsf{KBO}(s_i, t_i) \geqslant 0$$

where i denotes the least $1 \leqslant j \leqslant \min\{n, m\}$ with $s_i \neq t_i$.

Since the encoding of $\mathsf{KBO\text{-}PB}_k(s >_{\mathsf{kbo}} t)$ is explained in the example below here we just discuss the intended semantics of $\mathsf{KBO\text{-}PB}_k(s >_{\mathsf{kbo}'} t)$. In the first case where t is a variable there are no constraints on the weights and the precedence which means that the empty constraint is returned. In the other case the constraint expresses that whenever $\mathsf{KBO}'(s,t)$ is satisfied then either $f > g$ or both $f \sim g$ and $\mathsf{KBO}(s_i, t_i)$ must hold.

To get familiar with the encoding and to see why the definitions are a bit tricky consider the example below. For reasons of readability symbols occurring both in s and in t are removed immediately. This entails that the multiplication factor p should be lowered to

$$p = \sum_{a \in \mathcal{F} \cup \mathcal{V}} \max\{0, 2^k * (|t|_a - |s|_a)\},$$

which also is a lower bound on the left-hand side of constraint (2.1) if $\mathsf{KBO}(s,t)$ is zero because $\overline{w}_k(s) - \overline{w}_k(t) \geqslant -p$.

Example 2.24. Consider the TRS consisting of the rule

$$s = \mathsf{f}(\mathsf{g}(x), \mathsf{g}(\mathsf{g}(x))) \to \mathsf{f}(\mathsf{g}(\mathsf{g}(x)), x) = t.$$

The PB encoding $\mathsf{KBO\text{-}PB}_k(s >_{\mathsf{kbo}} t)$ then looks as follows:

$$-\mathsf{KBO}(s,t) + \overline{w}_k(\mathsf{g}) + \mathsf{KBO}'(s,t) \geqslant 0 \tag{2.2}$$
$$-2 * \mathsf{KBO}'(s,t) + 2 * X_{\mathsf{ff}} + Y_{\mathsf{ff}} + \mathsf{KBO}(\mathsf{g}(x), \mathsf{g}(\mathsf{g}(x))) \geqslant 0 \tag{2.3}$$
$$-(2^k + 1) * \mathsf{KBO}(\mathsf{g}(x), \mathsf{g}(\mathsf{g}(x))) - \overline{w}_k(\mathsf{g}) + \mathsf{KBO}'(\mathsf{g}(x), \mathsf{g}(\mathsf{g}(x))) \geqslant -2^k \tag{2.4}$$
$$-2 * \mathsf{KBO}'(\mathsf{g}(x), \mathsf{g}(\mathsf{g}(x))) + 2 * X_{\mathsf{gg}} + Y_{\mathsf{gg}} + \mathsf{KBO}(x, \mathsf{g}(x)) \geqslant 0 \tag{2.5}$$
$$\mathsf{KBO}(x, \mathsf{g}(x)) = 0 \tag{2.6}$$

Constraint (2.2) states that if $s >_{\mathsf{kbo}} t$ then either $\overline{w}_k(\mathsf{g}) > 0$ or $s >_{\mathsf{kbo}'} t$. Note that here the multiplication factor p is 0. Clearly the attentive reader would assign $\overline{w}_k(\mathsf{g}) = 1$ and termination of the TRS is shown. The encoding however is not so smart and performs the full recursive translation to PB. In (2.4) it is not possible to satisfy $s_1 = \mathsf{g}(x) >_{\mathsf{kbo}} \mathsf{g}(\mathsf{g}(x)) = t_1$ since the former is embedded in the latter. Nevertheless the constraint (2.4) must remain satisfiable because the TRS is KBO orientable. The trick is to introduce a hidden case analysis. The multiplication factor in front of the $\mathsf{KBO}(s_1, t_1)$ variable does that job. Whenever $s_1 >_{\mathsf{kbo}} t_1$ is needed then $\mathsf{KBO}(s_1, t_1)$ must evaluate to one. Then implicitly the constraint demands that $\overline{w}_k(s_1) > \overline{w}_k(t_1)$ or $\overline{w}_k(s_1) = \overline{w}_k(t_1)$ and $s_1 >_{\mathsf{kbo}'} t_1$ which reflects the definition of KBO. If $s_1 >_{\mathsf{kbo}} t_1$ need not be satisfied (e.g., because already $s >_{\mathsf{kbo}} t$ in (2.2) then the constraint holds in any case since the left-hand side in (2.4) never becomes smaller than -2^k because $\overline{w}_k(\mathsf{g}) < 2^k$.

The next definition expresses KBO in PB. The constraint $\mathsf{KBO\text{-}PB}_k(s >_{\mathsf{kbo}} t)$ demands that if $\mathsf{KBO}(s,t) = 1$ then $s >_{\mathsf{kbo}} t$. To ensure KBO orientation, for every rule $l \to r$ the constraint $\mathsf{KBO}(l,r) = 1$ is added. Note that without these additional constraints, the encoding would always be satisfiable, so also for TRSs that are not terminating.

Definition 2.25. Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$. The pseudo-boolean encoding $\mathsf{KBO\text{-}PB}_k(\mathcal{R})$ is the combination of $\mathsf{PREC\text{-}PB}(\mathcal{F})$, $\mathsf{ADM\text{-}PB}_k(\mathcal{F})$, and

$$\mathsf{KBO\text{-}PB}_k(l >_{\mathsf{kbo}} r);\ \mathsf{KBO}(l,r) = 1$$

for all $l \to r \in \mathcal{R}$.

Theorem 2.26. *Termination of $\mathcal{R}$ can be shown by KBO whenever the PBCs $\mathsf{KBO\text{-}PB}_k(\mathcal{R})$ are satisfiable.* □

The reverse holds for all $k \geqslant \lceil \log_2(B_{\mathcal{R}} + 1) \rceil$ (cf. Theorem 2.13).

2.4 Encodings with Dependency Pairs

One obvious and powerful extension of KBO is to integrate it in the dependency pair method.

In this section we reformulate the reduction pair processor with argument filterings and usable rules (Theorem 1.10) as a satisfiability problem over arithmetic constraints for specific orders. In Section 2.4.2 we address the embedding order and in Section 2.4.3 we address KBO, but first (Section 2.4.1) we explain how to represent argument filterings in propositional logic.

This encoding allows SAT and SMT as back-end. We note that for PB this approach does not seem so suitable since the resulting formula contains many disjunctions. The main motivation for using pseudo-boolean in the direct encoding (Section 2.3.2) was that it allows a concise implementation because the KBO constraints can easily be expressed in PB. This is no longer true when combining KBO with argument filterings. One could of course perform a Tseitin-like [82] transformation to PB but that would destroy the elegance of the approach. Why PB can still be advantageous is outlined in Section 2.6.

Before we actually start with encoding argument filterings we demonstrate a termination proof within the dependency pair setting by means of the following example. This example already shows that KBO gains much power by allowing argument filterings since they are capable of making duplicating rules non-duplicating.

Example 2.27. The TRS AG01/#3.1 consisting of the four rules

$$\begin{aligned} \mathsf{minus}(x,\mathsf{0}) &\to x & \mathsf{quot}(\mathsf{0},\mathsf{s}(y)) &\to \mathsf{0} \\ \mathsf{minus}(\mathsf{s}(x),\mathsf{s}(y)) &\to \mathsf{minus}(x,y) & \mathsf{quot}(\mathsf{s}(x),\mathsf{s}(y)) &\to \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x,y),\mathsf{s}(y))) \end{aligned}$$

gives rise to the dependency pairs

$$\begin{aligned} \mathsf{MINUS}(\mathsf{s}(x),\mathsf{s}(y)) &\to \mathsf{MINUS}(x,y) && (2.7)\\ \mathsf{QUOT}(\mathsf{s}(x),\mathsf{s}(y)) &\to \mathsf{MINUS}(x,y) \\ \mathsf{QUOT}(\mathsf{s}(x),\mathsf{s}(y)) &\to \mathsf{QUOT}(\mathsf{minus}(x,y),\mathsf{s}(y)). && (2.8) \end{aligned}$$

The dependency graph contains the two SCCs {2.7} and {2.8}. For the first one there are no usable rules and rule 2.7 can easily be handled by KBO. The second SCC is more challenging. To make rule 2.8 non-duplicating we take an argument filtering satisfying $\pi(\mathsf{QUOT}) = \pi(\mathsf{quot}) = \pi(\mathsf{minus}) = 1$ and $\pi(\mathsf{s}) = [1]$, producing $\mathsf{s}(x) \to x$ with filtered usable rules $x \to x$ and $\mathsf{s}(x) \to x$ originating from the rules defining minus. KBO can easily handle these rules.

2.4.1 Representing Argument Filterings

Within this subsection $\mathcal{F}$ is considered as a signature over a DP problem $(\mathcal{P}, \mathcal{R})$.

Definition 2.28. For a signature $\mathcal{F}$ we denote by $\pi_{\mathcal{F}}$ the set of propositional variables $\{\pi_f \mid f \in \mathcal{F}\} \cup \{\pi_f^i \mid f \in \mathcal{F} \text{ and } 1 \leqslant i \leqslant \text{arity}(f)\}$. Let π be an argument filtering for $\mathcal{F}$. The induced assignment α_π is defined as follows:

$$\alpha_\pi(\pi_f) = \begin{cases} 1 & \text{if } \pi(f) = [i_1, \ldots, i_m] \\ 0 & \text{if } \pi(f) = i \end{cases} \quad \text{and} \quad \alpha_\pi(\pi_f^i) = \begin{cases} 1 & \text{if } i \in \pi(f) \\ 0 & \text{if } i \notin \pi(f) \end{cases}$$

for all n-ary function symbols $f \in \mathcal{F}$ and $i \in \{1, \ldots, n\}$.

Definition 2.29. An assignment α for $\pi_{\mathcal{F}}$ is said to be *argument filtering consistent* if for every n-ary function symbol $f \in \mathcal{F}$ such that $\alpha \not\vDash \pi_f$ there is a unique $i \in \{1, \dots, n\}$ such that $\alpha \vDash \pi_f^i$.

It is easy to see that α_π is argument filtering consistent.

Definition 2.30. The propositional formula $\mathsf{AF}^\pi(\mathcal{F})$ is defined as $\bigwedge_{f \in \mathcal{F}} \mathsf{AF}^\pi(f)$ with

$$\mathsf{AF}^\pi(f) = \pi_f \vee \bigvee_{i=1}^{\operatorname{arity}(f)} \big(\pi_f^i \wedge \bigwedge_{j \neq i} \neg \pi_f^j\big).$$

Lemma 2.31. *An assignment α for $\pi_{\mathcal{F}}$ is argument filtering consistent if and only if $\alpha \vDash \mathsf{AF}^\pi(\mathcal{F})$.* □

Definition 2.32. Let α be an argument filtering consistent assignment for $\pi_{\mathcal{F}}$. The argument filtering π_α is defined as follows: $\pi_\alpha(f) = [i \mid \alpha \vDash \pi_f^i]$ if $\alpha \vDash \pi_f$ and $\pi_\alpha(f) = i$ if $\alpha \not\vDash \pi_f$ and $\alpha \vDash \pi_f^i$, for all function symbols $f \in \mathcal{F}$.

Example 2.33. Consider the signature from Example 2.27. The assignment α only satisfying the variables π_{MINUS}, π^2_{MINUS}, π_{minus}, π^2_{QUOT}, π^1_{quot}, π_{0}, and π_{s} is argument filtering consistent. The induced argument filtering π_α consists of $\pi_\alpha(\mathsf{MINUS}) = [2]$, $\pi_\alpha(\mathsf{minus}) = \pi_\alpha(\mathsf{0}) = \pi_\alpha(\mathsf{s}) = [\,]$, $\pi_\alpha(\mathsf{QUOT}) = 2$, and $\pi_\alpha(\mathsf{quot}) = 1$.

Corresponding to the definition of $\mathcal{U}_\pi(\mathcal{P}, \mathcal{R})$ and Theorem 1.10 from page 9 we encode $\pi(\mathcal{U}_\pi(\mathcal{P}, \mathcal{R}) \cup \mathcal{P}) \subseteq \gtrsim$ as the conjunction of

$$\bigwedge_{l \to r \in \mathcal{P}} \big(U_{\operatorname{root}(l)} \wedge l \gtrsim^\pi r\big) \wedge \bigwedge_{l \to r \in \mathcal{R}} \big(U_{\operatorname{root}(l)} \to l \gtrsim^\pi r\big)$$

and

$$\bigwedge_{l \to r \,\in\, \mathcal{R} \cup \mathcal{P}} \Bigg(U_{\operatorname{root}(l)} \to \bigwedge_{\substack{p \in \mathcal{P}\mathrm{os}_{\mathcal{F}}(r) \\ \operatorname{root}(r|_p) \text{ is defined}}} \Bigg(\bigwedge_{q,\, i \colon iq \leqslant p} \pi^i_{\operatorname{root}(r|_q)} \to U_{\operatorname{root}(r|_p)} \Bigg)\Bigg).$$

Here $l \gtrsim^\pi r$ abbreviates $\pi(l) \gtrsim \pi(r)$. Furthermore, U_f is a new propositional variable for every defined and every dependency pair symbol f. If U_f evaluates to true, then rules of the form $f(\dots) \to r$ must be oriented. In the formula above the first conjunct expresses that all rules from $\mathcal{P}$ must be oriented by $\gtrsim^\pi$. The second conjunct expresses that if a rule is usable, then it must be compatible with $\gtrsim^\pi$ whereas the third conjunct computes the usable rules with respect to an argument filtering (if a rule is usable, then defined symbols—occurring on positions that are not deleted by the argument filtering—in its right-hand side also give rise to usable rules). The relation $\gtrsim^\pi$ can be replaced by an encoding of $\gtrsim$ (the weak part of a reduction pair) that incorporates argument filterings π. The above formula is a crucial ingredient for implementing the DP processor from Theorem 1.10 and is abbreviated by $\mathsf{U}(\mathcal{P}, \mathcal{R}, \gtrsim^\pi)$.

2.4.2 Embedding

To reformulate Theorem 1.10 as a satisfaction problem, we must fix a reduction pair, incorporate argument filterings, and encode the combination in propositional logic. Next we take the reduction pair $(\trianglerighteq_{\mathsf{emb}}, \triangleright_{\mathsf{emb}})$ corresponding to the embedding order. Because embedding has no parameters it allows for a transparent translation of the constraints $\pi(\mathcal{U}_\pi(\mathcal{P}, \mathcal{R}) \cup \mathcal{P}) \subseteq \gtrsim$ and $\pi(\mathcal{P}) \cap > \neq \varnothing$ needed to implement Theorem 1.10. In Section 2.4.3 we consider KBO, which is a bit more challenging.

Definition 2.34. The *embedding* relation $\trianglelefteq_{\mathsf{emb}}$ is defined on terms as follows: $s \trianglelefteq_{\mathsf{emb}} t$ if $s = t$ or $t = f(t_1, \ldots, t_n)$ and either $s \trianglelefteq_{\mathsf{emb}} t_i$ for some i or $s = f(s_1, \ldots, s_n)$ and $s_i \trianglelefteq_{\mathsf{emb}} t_i$ for all i. The strict part is denoted by $\triangleleft_{\mathsf{emb}}$. The converse relations are denoted by $\trianglerighteq_{\mathsf{emb}}$ and $\triangleright_{\mathsf{emb}}$.

In the following we define propositional formulas $s \triangleright^\pi_{\mathsf{emb}} t$ and $s \trianglerighteq^\pi_{\mathsf{emb}} t$ which, in conjunction with $\mathsf{AF}^\pi(\mathcal{F})$, represent all argument filterings π that satisfy $\pi_\alpha(s) \triangleright_{\mathsf{emb}} \pi_\alpha(t)$ and $\pi_\alpha(s) \trianglerighteq_{\mathsf{emb}} \pi_\alpha(t)$. We start with defining a formula $s =^\pi t$ that represents all argument filterings which make s and t equal. (In the remainder of this chapter we assume that $\wedge$ binds stronger than $\vee$.)

Definition 2.35. Let s and t be terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We define a propositional formula $s =^\pi t$ by induction on s and t. If $s \in \mathcal{V}$ then

$$s =^\pi t \;=\; \begin{cases} \top & \text{if } s = t, \\ \bot & \text{if } t \in \mathcal{V} \text{ and } s \neq t, \\ \neg\pi_g \wedge \bigvee_{j=1}^{m} \big(\pi_g^j \wedge s =^\pi t_j\big) & \text{if } t = g(t_1, \ldots, t_m). \end{cases}$$

Let $s = f(s_1, \ldots, s_n)$. If $t \in \mathcal{V}$ then

$$s =^\pi t \;=\; \neg\pi_f \wedge \bigvee_{i=1}^{n} \big(\pi_f^i \wedge s_i =^\pi t\big).$$

If $t = g(t_1, \ldots, t_m)$ with $f \neq g$ then

$$s =^\pi t \;=\; \neg\pi_f \wedge \bigvee_{i=1}^{n} \big(\pi_f^i \wedge s_i =^\pi t\big) \vee \neg\pi_g \wedge \bigvee_{j=1}^{m} \big(\pi_g^j \wedge s =^\pi t_j\big).$$

Finally, if $t = f(t_1, \ldots, t_n)$ then

$$s =^\pi t \;=\; \neg\pi_f \wedge \bigvee_{i=1}^{n} \big(\pi_f^i \wedge s_i =^\pi t_i\big) \vee \pi_f \wedge \bigwedge_{i=1}^{n} \big(\pi_f^i \to s_i =^\pi t_i\big).$$

For readability we present a translation related as close as possible to the definition of argument filterings. In an implementation one should minimize the formulas, e.g., the last formula can be expressed more concisely as

$$s =^\pi t \;=\; \bigwedge_{i=1}^{n} \big(\pi_f^i \to s_i =^\pi t_i\big)$$

since we know that $\mathsf{AF}^\pi(\mathcal{F})$ must hold anyway.

Definition 2.36. Let s and t be terms in $\mathcal{T}(\mathcal{F},\mathcal{V})$. We define propositional formulas $s \rhd^\pi_{\mathsf{emb}} t$ and $s \trianglerighteq^\pi_{\mathsf{emb}} t = s \rhd^\pi_{\mathsf{emb}} t \vee s =^\pi t$ by induction on s and t. If $s \in \mathcal{V}$ then $s \rhd^\pi_{\mathsf{emb}} t \;=\; \bot$. Let $s = f(s_1,\dots,s_n)$. If $t \in \mathcal{V}$ then

$$s \rhd^\pi_{\mathsf{emb}} t \;=\; \pi_f \wedge \bigvee_{i=1}^{n} \big(\pi_f^i \wedge s_i \trianglerighteq^\pi_{\mathsf{emb}} t\big) \vee \neg\pi_f \wedge \bigvee_{i=1}^{n} \big(\pi_f^i \wedge s_i \rhd^\pi_{\mathsf{emb}} t\big).$$

If $t = g(t_1,\dots,t_m)$ with $f \neq g$ then $s \rhd^\pi_{\mathsf{emb}} t$ is the disjunction of

$$\pi_f \wedge \Big(\pi_g \wedge \bigvee_{i=1}^{n} \big(\pi_f^i \wedge s_i \trianglerighteq^\pi_{\mathsf{emb}} t\big) \vee \neg\pi_g \wedge \bigvee_{j=1}^{m} \big(\pi_g^j \wedge s \rhd^\pi_{\mathsf{emb}} t_j\big)\Big)$$

and $\neg\pi_f \wedge \bigvee_{i=1}^{n} \big(\pi_f^i \wedge s_i \rhd^\pi_{\mathsf{emb}} t\big)$. Finally, if $t = f(t_1,\dots,t_n)$ then

$$\begin{aligned} s \rhd^\pi_{\mathsf{emb}} t \;=\; \pi_f \wedge \Big(& \bigvee_{i=1}^{n} \big(\pi_f^i \wedge s_i \trianglerighteq^\pi_{\mathsf{emb}} t\big) \vee \bigwedge_{i=1}^{n} \big(\pi_f^i \to s_i \trianglerighteq^\pi_{\mathsf{emb}} t_i\big) \wedge \\ & \bigvee_{i=1}^{n} \big(\pi_f^i \wedge s_i \rhd^\pi_{\mathsf{emb}} t_i\big)\Big) \vee \neg\pi_f \wedge \bigvee_{i=1}^{n} \big(\pi_f^i \wedge s_i \rhd^\pi_{\mathsf{emb}} t_i\big). \end{aligned}$$

The formula $s \rhd^\pi_{\mathsf{emb}} t \wedge \mathsf{AF}^\pi(\mathcal{F})$ is satisfiable if and only if there exists an argument filtering π such that $\pi(s) \rhd_{\mathsf{emb}} \pi(t)$. Actually the statement is even stronger, since $s \rhd^\pi_{\mathsf{emb}} t \wedge \mathsf{AF}^\pi(\mathcal{F})$ encodes *all* argument filterings π that satisfy $\pi(s) \rhd_{\mathsf{emb}} \pi(t)$. Analogous statements hold for $s =^\pi t \wedge \mathsf{AF}^\pi(\mathcal{F})$ and $s \trianglerighteq^\pi_{\mathsf{emb}} t \wedge \mathsf{AF}^\pi(\mathcal{F})$.

Lemma 2.37. *Let s and $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$. If α is an assignment for $\pi_\mathcal{F}$ such that $\alpha \vDash s \rhd^\pi_{\mathsf{emb}} t \wedge \mathsf{AF}^\pi(\mathcal{F})$ then $\pi_\alpha(s) \rhd_{\mathsf{emb}} \pi_\alpha(t)$. If π is an argument filtering such that $\pi(s) \rhd_{\mathsf{emb}} \pi(t)$ then $\alpha_\pi \vDash s \rhd^\pi_{\mathsf{emb}} t \wedge \mathsf{AF}^\pi(\mathcal{F})$.* □

We conclude this subsection by stating the propositional formulation of the DP processor of Theorem 1.10 specialized to embedding.

Theorem 2.38. *Let $(\mathcal{P},\mathcal{R})$ be a DP problem over a signature $\mathcal{F}$. The formula*

$$\mathsf{U}(\mathcal{P},\mathcal{R},\trianglerighteq^\pi_{\mathsf{emb}}) \wedge \mathsf{AF}^\pi(\mathcal{F}) \wedge \bigvee_{s\to t\in\mathcal{P}} s \rhd^\pi_{\mathsf{emb}} t$$

is satisfiable if and only if there exists an argument filtering π such that the constraints $\pi(\mathcal{U}_\pi(\mathcal{P},\mathcal{R}) \cup \mathcal{P}) \subseteq \trianglerighteq_{\mathsf{emb}}$ and $\pi(\mathcal{P}) \cap \rhd_{\mathsf{emb}} \neq \varnothing$ hold.[5] □

2.4.3 Knuth-Bendix Order

Our aim is to define a formula

$$s >^\pi_{\mathsf{kbo}} t \wedge \mathsf{AF}^\pi(\mathcal{F}) \wedge \mathsf{ADM}^\pi(\mathcal{F})$$

[5] Independently, in [11] a similar encoding is presented for LPO.

that is satisfiable if and only if there exist an argument filtering π and a precedence $>$ such that $\pi(s) >_{\mathsf{kbo}} \pi(t)$. The conjunct $\mathsf{ADM}^{\pi}(\mathcal{F})$ takes care of the admissibility condition.

Below we define the conjunct $s >^{\pi}_{\mathsf{kbo}} t$. The basic idea is to adapt $s \rhd^{\pi}_{\mathsf{emb}} t$ by incorporating the recursive definition of $>_{\mathsf{kbo}}$. First we propose a formula that expresses that after applying the argument filtering no variables are duplicated.

Definition 2.39. The formula $\mathsf{ND}^{\pi}(s,t)$ is defined as follows:

$$\mathsf{ND}^{\pi}(s,t) \;=\; \bigwedge_{x \,\in\, \mathcal{V}\mathsf{ar}(t)} |s,\top|_x \geqslant |t,\top|_x$$

with

$$|s,\varphi|_x \;=\; \begin{cases} \varphi \;?\; 1 : 0 & \text{if } s = x, \\ 0 & \text{if } s \in \mathcal{V} \text{ and } s \neq x, \\ \displaystyle\sum_{i=1}^{n} |s_i, \varphi \wedge \pi_f^i|_x & \text{if } s = f(s_1,\ldots,s_n). \end{cases}$$

The idea behind the recursive definition of $|s,\varphi|_x$ is to collect the constraints under which a variable is preserved by the argument filtering. If those constraints are satisfied they correspond to an occurrence of the variable. Adding the constraints yields the number of variables which survive the argument filtering.

Example 2.40. Consider the rule $l = \mathsf{MINUS}(\mathsf{s}(x),\mathsf{s}(y)) \to \mathsf{MINUS}(x,y) = r$. Then the formula $\mathsf{ND}^{\pi}(l,r)$ evaluates (modulo simplifications) to

$$(\pi^1_{\mathsf{MINUS}} \wedge \pi^1_{\mathsf{s}} \;?\; 1 : 0) \geqslant (\pi^1_{\mathsf{MINUS}} \;?\; 1 : 0) \wedge (\pi^2_{\mathsf{MINUS}} \wedge \pi^1_{\mathsf{s}} \;?\; 1 : 0) \geqslant (\pi^2_{\mathsf{MINUS}} \;?\; 1 : 0)$$

where the first (second) conjunct expresses non-duplication of variable x (y). Informally, the formula states that whenever an argument filtering π keeps the first (or second) argument of MINUS, then it must also keep the argument of s.

Next we give a formula that computes the weight of a term after an argument filtering has been applied.

Definition 2.41. We define $\mathrm{w}^{\pi}(t)$ as $\mathrm{w}'_{\pi}(t,\top)$ with

$$\mathrm{w}'_{\pi}(t,\varphi) \;=\; \begin{cases} \varphi \;?\; \mathrm{w}_f : 0 & \text{if } t \in \mathcal{V}, \\ (\pi_f \wedge \varphi \;?\; \mathrm{w}_f : 0) + \displaystyle\sum_{i=1}^{n} \mathrm{w}'_{\pi}(t_i, \pi^i_f \wedge \varphi) & \text{if } t = f(t_1,\ldots,t_n). \end{cases}$$

Definition 2.42. Let s and t be terms. We define propositional formulas

$$s >^{\pi}_{\mathsf{kbo}} t \;=\; \mathsf{ND}^{\pi}(s,t) \wedge \big(\mathrm{w}^{\pi}(s) > \mathrm{w}^{\pi}(t) \;\vee\; \mathrm{w}^{\pi}(s) = \mathrm{w}^{\pi}(t) \;\wedge\; s >^{\pi}_{\mathsf{kbo}'} t\big)$$

and

$$s \geqslant^{\pi}_{\mathsf{kbo}} t \;=\; s >^{\pi}_{\mathsf{kbo}} t \;\vee\; s =^{\pi} t$$

with $s >^{\pi}_{\mathsf{kbo}'} t$ inductively defined as follows. If $s \in \mathcal{V}$ then $s >^{\pi}_{\mathsf{kbo}'} t \;=\; \bot$. Let $s = f(s_1, \ldots, s_n)$. If $t \in \mathcal{V}$ then $s >^{\pi}_{\mathsf{kbo}'} t \;=\; s \rhd^{\pi}_{\mathsf{emb}} t$. If $t = g(t_1, \ldots, t_m)$ with $f \neq g$ then

$$s >^{\pi}_{\mathsf{kbo}'} t \;=\; \pi_f \wedge \pi_g \wedge\; \mathsf{p}_f > \mathsf{p}_g \vee \\ \neg\pi_g \wedge \bigvee_{j=1}^{m} (\pi_g^j \wedge s >^{\pi}_{\mathsf{kbo}} t_j) \vee \neg\pi_f \wedge \bigvee_{i=1}^{n} (\pi_f^i \wedge s_i >^{\pi}_{\mathsf{kbo}} t).$$

Finally, if $t = f(t_1, \ldots, t_n)$ then

$$s >^{\pi}_{\mathsf{kbo}'} t \;=\; \pi_f \wedge \langle s_1, \ldots, s_n\rangle >^{\pi,f}_{\mathsf{kbo}} \langle t_1, \ldots, t_n\rangle \vee \neg\pi_f \wedge \bigvee_{i=1}^{n} (\pi_f^i \wedge s_i >^{\pi}_{\mathsf{kbo}} t_i).$$

Here $\langle s_1, \ldots, s_n\rangle >^{\pi,f}_{\mathsf{kbo}} \langle t_1, \ldots, t_n\rangle$ is defined as $\bot$ if $n = 0$ and as

$$\pi_f^1 \wedge s_1 >^{\pi}_{\mathsf{kbo}} t_1 \vee (\pi_f^1 \to s_1 =^{\pi} t_1) \wedge \langle s_2, \ldots, s_n\rangle >^{\pi,f}_{\mathsf{kbo}} \langle t_2, \ldots, t_n\rangle$$

if $n > 0$.

Note that $s >^{\pi}_{\mathsf{kbo}'} t$ corresponds to the definition of KBO in the case of equal weights (cf. Definition 2.1 case (b)(2)). The equation $s >^{\pi}_{\mathsf{kbo}'} t = s \rhd^{\pi}_{\mathsf{emb}} t$ for $t \in \mathcal{V}$ might look peculiar but can be explained by the admissibility condition (encoded below) and the fact that $\pi(s)$ and $\pi(t) = t$ are assumed to have equal weight. The formula $\mathsf{ADM}^{\pi}(\mathcal{F})$ from the next definition is satisfiable if and only if the encoded weight function is admissible in the presence of an argument filtering. Here $\mathsf{C}^{\pi}(f)$ and $\mathsf{U}^{\pi}(f)$ express that $\pi(f)$ is constant and unary, respectively.

Definition 2.43. For a signature $\mathcal{F}$, let $\mathsf{ADM}^{\pi}(\mathcal{F})$ be the formula

$$\mathrm{w}_0 > 0 \wedge \bigwedge_{f \in \mathcal{F}} \big(\mathsf{C}^{\pi}(f) \to \mathrm{w}_f \geqslant \mathrm{w}_0\big) \wedge \\ \bigwedge_{f \in \mathcal{F}} \big(\mathsf{U}^{\pi}(f) \wedge \mathrm{w}_f = 0 \;\to\; \bigwedge_{g \in \mathcal{F}} (\pi_g \to \mathsf{p}_f \geqslant \mathsf{p}_g)\big)$$

with $\mathsf{C}^{\pi}(f) = \pi_f \wedge \bigwedge_{i=1}^{\mathrm{arity}(f)} \neg\pi_f^i$ and $\mathsf{U}^{\pi}(f) = \pi_f \wedge \bigvee_{i=1}^{\mathrm{arity}(f)} \big(\pi_f^i \wedge \bigwedge_{i \neq j} \neg\pi_f^j\big)$.

Theorem 2.44. *Let $(\mathcal{P}, \mathcal{R})$ be a DP problem over a signature $\mathcal{F}$. The formula*

$$\mathsf{U}(\mathcal{P}, \mathcal{R}, \geqslant^{\pi}_{\mathsf{kbo}}) \wedge \mathsf{ADM}^{\pi}(\mathcal{F}) \wedge \mathsf{AF}^{\pi}(\mathcal{F}) \wedge \bigvee_{s \to t \in \mathcal{P}} s >^{\pi}_{\mathsf{kbo}} t$$

is satisfiable if and only if there are an argument filtering π, a precedence $>$, and an admissible weight function (w, w_0) such that $\pi(\mathcal{U}_{\pi}(\mathcal{P}, \mathcal{R}) \cup \mathcal{P}) \subseteq \geqslant_{\mathsf{kbo}}$ and $\pi(\mathcal{P}) \cap >_{\mathsf{kbo}} \neq \varnothing$. □

From a satisfying assignment one can read off the argument filtering, the precedence, and the weight function. We omit the straightforward details. When solving the constraints from Theorem 2.44 the SMT back-end always gives a complete implementation whereas the SAT approach is only complete for sufficiently large weights. However, again due to the results of Section 2.2 upper bounds on weights can always be computed and experiments (Section 2.5.3) show that in practice small weights suffice.

2.5 Experiments

Below we compare our implementations of KBO, sat, pbc, and smt (on top of T$_{\text{T}}$T$_2$) with the ones of T$_{\text{T}}$T [39], a special version of AProVE [30] provided by the authors, and an implementation dkm (also on top of T$_{\text{T}}$T$_2$) as proposed by Dick *et al.* in [17]. T$_{\text{T}}$T and AProVE admit only strict precedences. Both implement the algorithm of Korovin and Voronkov [53] together with techniques from [17]. For two of our approaches (sat and pbc) KBO orientability amounts to finding a satisfying assignment for a propositional formula whereas the smt approach is based on linear programming. The other tools find a solution by solving a system of homogeneous linear inequations which also amounts to linear programming. Although this problem is known to be decidable in polynomial time [47, 46] in practice algorithms with exponential (worst-case) time complexity such as the simplex method [15] perform much better. Dick *et al.* [17] prefer the method of complete description over simplex due to its support for incrementality. This shows that although computing a KBO for a given TRS can be done in polynomial time, none of the existing tools does so.

Concerning optimizations, when computing weights of terms symbols occurring on both sides of rules are ignored. Furthermore the encoding of KBO (in the direct setting) is only computed if a test for embedding fails. Keeping the encoding in a cache allows to re-use precomputed formulas which drastically reduces encoding time. For all data given in the following tables addition in SAT (Definition 6.2) takes overflows into account, i.e., adding two k-bit numbers results in a $(k+1)$-bit number.

2.5.1 Results for TRSs

As addressed earlier for the SAT and PB back-ends one has to fix the number k of bits which is used to represent natural numbers in binary representation. The actual choice is specified as argument to sat (pbc). Note that a rather small k is sufficient to handle all potential systems from TPDB which makes Theorems 2.20 and 2.26 powerful in practice also for the SAT and PB back-ends. As already indicated in Example 2.4 there does not exist a uniform upper bound on k but for every given TRS one can compute such a k according to Theorem 2.13.

In Table 2.1 we compare different approaches dealing with KBO. The columns labeled yes indicate the number of successful proofs while t/o (timeout) counts how often execution was killed since the tool did not produce an answer within 60 seconds. The column total time is measured in seconds. The left part of

Table 2.1: KBO for 1391 TRSs

	strict precedence			quasi-precedence		
method(#bits)	yes	total time	t/o	yes	total time	t/o
sat/pbc(2)	104/104	24.7/174.9	0/0	105/104	23.7/247.9	0/1
sat/pbc(3)	106/106	24.1/180.1	0/0	107/107	24.6/223.4	0/0
sat/pbc(4)	107/107	28.0/181.4	0/0	108/108	26.3/229.7	0/0
sat/pbc(10)	107/107	257.3/267.2	3/1	108/107	261.8/335.6	3/2
$\mathsf{smt_i}$	107	24.0	0	108	19.9	0
$\mathsf{smt_r}$	107	23.9	0	108	19.5	0
AProVE	101	1946.0	18			
T$_\mathsf{T}$T	101	335.0	1			
T$_\mathsf{T}$T(simplex)	105	360.5	4			
dkm	99	801.4	13			
dkm'	102	436.1	7			

Table 2.1 summarizes the results for strict precedences. Interestingly, already $k = 4$ suffices to prove the maximum number of systems terminating. The applicative[6] TRS higher-order/AProVE/HO/ReverseLastInit consisting of the rules

$$\begin{aligned}
\mathsf{compose}\, f\, g\, x &\to g\,(f\,x) & \mathsf{hd}\,(\mathsf{cons}\, x\, xs) &\to x\\
\mathsf{reverse}\, l &\to \mathsf{reverse2}\, l\, \mathsf{nil} & \mathsf{tl}\,(\mathsf{cons}\, x\, xs) &\to xs\\
\mathsf{reverse2}\,(\mathsf{cons}\, x\, xs)\, l &\to \mathsf{reverse2}\, xs\,(\mathsf{cons}\, x\, l) & \mathsf{reverse2}\,\mathsf{nil}\, l &\to l\\
\mathsf{last} &\to \mathsf{compose}\,\mathsf{hd}\,\mathsf{reverse}\\
\mathsf{init} &\to \mathsf{compose}\,\mathsf{reverse}\,(\mathsf{compose}\,\mathsf{tl}\,\mathsf{reverse})
\end{aligned}$$

can only be handled by sat and pbc with $k \geqslant 4$. The constant reverse needs at least weight three and thus nine is the smallest weight for the constant init. The SMT approach does not need to represent numbers in binary and consequently there are no bit-restrictions on the weights. Furthermore, this back-end allows to choose the real numbers as domain for the weights. However, the SMT solver we use (Yices) can only deal with rationals. The index for smt indicates if integers ($\mathsf{smt_i}$) or rationals ($\mathsf{smt_r}$) are employed.

Since T$_\mathsf{T}$T and AProVE implement the slightly stronger KBO definition of [53] they can prove two TRSs (various/27 and TRCSR/Ex9_Luc06_GM) terminating which cannot be handled by our methods. (We did not investigate if one can specialize the encodings to also capture these systems but are convinced that this is in principle possible.) On the other hand T$_\mathsf{T}$T gives up on HM/t000 (and six more TRSs that derive from context sensitive rewriting) which specifies addition for natural numbers in decimal notation (using 104 rewrite rules). The problem is not the time limit but at some point the algorithm detects that it will require too many resources. To prevent running out of memory, the computation is terminated and a "don't know" result is reported. AProVE does not cut off execution and consequently for this system (and 17 others) no result is obtained within 60 seconds. Also dkm fails on HM/t000 (t/o) whereas

[6] In applicative notation the binary function symbol app is written as juxtaposition and left-associative. E.g., the term $\mathsf{hd}\,(\mathsf{cons}\, x\, xs)$ represents $\mathsf{app}(\mathsf{hd}, \mathsf{app}(\mathsf{app}(\mathsf{cons}, x), xs))$.

Table 2.2: KBO for 732 SRSs

	strict precedence			quasi-precedence		
method(#bits)	yes	total time	t/o	yes	total time	t/o
sat/pbc(3)	24/24	10.0/8.0	0/0	24/24	11.1/9.1	0/0
sat/pbc(4)	30/30	10.8/8.4	0/0	30/30	13.6/9.5	0/0
sat/pbc(6)	33/33	13.7/9.5	0/0	33/33	17.1/10.8	0/0
sat/pbc(8)	33/33	16.7/10.1	0/0	33/33	20.3/13.2	0/0
$\mathsf{smt_i}$	33	9.4	0	33	9.0	0
$\mathsf{smt_r}$	33	9.6	0	33	8.9	0
AProVE	30	676.6	4			
TTT	30	44.3	0			
TTT(simplex)	33	21.2	0			
dkm	29	366.9	6			
dkm'	30	255.8	4			

for none of our approaches this system seems to pose a problem; sat(4), pbc(4), $\mathsf{smt_i}$, and $\mathsf{smt_r}$ succeed within 0.19, 0.16, 0.03, and 0.03 seconds. The algorithm dkm produces large sparse matrices during execution for some systems (e.g. for HM/t000 after 30 seconds 1 GB of memory is used). We developed an OCaml module for sparse matrices which could drastically reduce memory usage (e.g. for HM/t000 after 30 seconds only 50 MB). Nevertheless this effort just slightly improves the data for dkm. On the whole database the number of successful proofs did not increase and execution time decreases just slightly. Another issue that increased performance of dkm much more was *sorting* the set of equations in order to keep the internal data-structure (the matrices S_i^A) for MCD much smaller. This allowed us to prove various/21 in 0.03 seconds whereas it was intractable for this method beforehand (out-of-memory after eight minutes). The idea of sorting somehow contradicts the claim in [17] that MCD is preferable over the simplex method [15] due to its incremental nature. Our tests showed that restarting MCD (with sorted inequalities) whenever some new inequalities are added performs better than just incrementally adding one inequality after the other. Table 2.1 shows that by sorting (dkm') the method can prove three additional TRSs while the execution time drops by a factor of two. We also varied (within TTT) the back-end for solving linear inequations. While the standard implementation of TTT uses MCD, the special version TTT(simplex) provided by Nao Hirokawa implements the first phase of [15]. Again the results contradict the claim in [17] that MCD is better suited than the simplex method.

As can be seen in the right part of Table 2.1, by admitting quasi-precedences one additional TRS (SK90/2.42, Example 2.3) can be proved terminating.

2.5.2 Results for SRSs

For SRSs we have similar results, as can be inferred from Table 2.2. The main difference is the larger number of bits needed for the propositional representation of the weights. The maximum number of SRSs is proved terminating with $k \geqslant 6$. Generally speaking TTT performs better on SRSs than on TRSs con-

Table 2.3: KBO with dependency pairs for 1391 TRSs/732 SRSs

	TRSs			SRSs		
method(#bits)	yes	total time	t/o	yes	total time	t/o
sat(2)	488	1912.9	16	45	117.4	0
sat(3)	491	2440.6	15	48	261.4	0
sat(4)	491	4915.5	36	55	524.7	1
sat(5)	489	8570.8	85	56	1098.8	3
sat(6)	488	11870.0	140	57	1820.4	7
sat(10)	487	17820.6	240	55	6388.7	43
smt_i	488	2353.2	22	57	119.1	1
smt_r	489	1862.9	18	57	105.6	1
AProVE	445	15727.8	235	37	2873.1	37
TTT	323	24095.2	370	27	2644.9	36

cerning KBO because it can handle all systems within the time limit. However, again our experiments reveal that the implementation of TTT is not complete, i.e., it proves termination of 30 SRSs only whereas our implementations succeed on 33 SRSs. The three SRSs that make up the difference (Trafo/dup11, Zantema/z069, Zantema/z070) derive from algebra (polyhedral groups). Also AProVE and dkm are unable to handle these systems (t/o) while the simplex version of TTT performs well. Admitting quasi-precedences does not allow to prove additional SRSs from TPDB terminating by KBO.

2.5.3 Results with Dependency Pairs

Apart from the concepts explained in Sections 1.2 and 2.4 for the results in this subsection the dependency graph refinements presented in [32, 38] and usable rules like in [32] have been considered.

In combination with the dependency pair setting KBO gains much power compared to the direct approach. One reason is that by allowing argument filterings duplicating systems may become non-duplicating and consequently this (severe) restriction is eased.

We implemented the encoding from Section 2.4 in TTT_2 yielding implementations for sat and smt. The following strategy skeleton is used to call TTT_2 in this setting: `var | dp;edg;(sccs;ur;kbo -dp -ur □)*`. To obtain an executable strategy, □ must be replaced by concrete flags, e.g., `-sat -ib 2`. More information on TTT_2's strategy language is available in Appendix A and the exact strategies that have been used to produce the tables are listed in Section A.4.

Table 2.3[7] shows that the smt encoding is fastest and for rational weights the solving time is drastically reduced compared to the integer case. Interestingly even if the weights are allowed to take rational values Yices returns integer solutions for almost all systems, see Section 2.6 for details. Although smt_r misses proving two systems compared to sat(3) it remains the optimal choice since no

[7] We stress that TTT has a weaker implementation of the dependency pair framework and consequently the results cannot directly be compared.

parameters have to be chosen to call the method. Furthermore the two missing systems can be handled by slightly increasing the timeout, i.e., smt_r is successful on TRCSR/PALINDROME_complete-noand_FR within 86.4 seconds and spends 151.8 seconds on TRCSR/PALINDROME_complete-noand_Z. For the SRS category SMT is the clear winner since it is by far the fastest while yielding maximal termination proving power as demonstrated in the right part of Table 2.3.

2.6 Assessment

In this section we compare the three new approaches presented in this chapter. Let us start with the most important measurements: power and run time. Here smt is the clear winner. In [89] smt was not considered and pbc performed best. Since here a different database is employed the results look slightly worse for PB. On most examples pbc still outperforms sat but on a few systems (transformations from context-sensitive rewriting) pbc is not efficient at all. But pbc still scales better when using more bits (cf. Table 2.1). Furthermore, the pseudo-boolean approach is less implementation work since additions are performed by the solver and also the transformation to CNF is not necessary. However it is not straightforward to extend the PB encoding by argument filterings. Of course smt combines the benefits of both approaches. The encoding is straightforward, little effort to implement, and efficient.

But an advantage of the pseudo-boolean approach is the option of a *goal function* which should be minimized while preserving satisfiability of the constraints. Although the usage of such a goal function is not of computational interest it is useful for generating easily human readable proofs. We experimented with functions reducing the comparisons in the precedence and minimizing weights for function symbols. Concerning the former we detected that using two (three, four, ten) bits to encode weights of function symbols 49 (56, 60, 60) TRSs can be proved terminating with empty precedence. The latter has the advantage that one obtains minimal weights in KBO proofs which is nicely illustrated on the SRS Zantema/z113 consisting of the rules

$$\begin{array}{lll} 11 \to 43 & 33 \to 56 & 55 \to 62 \\ 12 \to 21 & 22 \to 111 & 34 \to 11 \\ 44 \to 3 & 56 \to 12 & 66 \to 21. \end{array}$$

T$_\mathsf{T}$T and AProVE produce the proof

$$\begin{array}{lll} w(1) = 32471712256 & w(2) = 48725750528 & w(3) = 43247130624 \\ w(4) = 21696293888 & w(5) = 44731872512 & w(6) = 40598731520 \\ 3 > 1 > 2 & & 1 > 4 \end{array}$$

whereas pbc(6) yields

$$\begin{array}{lll} w(1) = 31 & w(2) = 47 & w(3) = 41 \\ w(4) = 21 & w(5) = 43 & w(6) = 39 \\ 3 > 1 > 2 & 3 > 5 > 6 > 2 & 1 > 4. \end{array}$$

So for the first time it became clear that these large numbers are not needed to prove KBO orientability of the system Zantema/z113. To some extent it is clear that AProVE and T$_\text{T}$T produce such large weights since these implementations are based on the work in [17] which always ensures minimal precedences. Surprisingly our dkm' implementation produced a proof for various/21 with minimal (since empty) precedence and weight function $w(+) = 12$, $w(\mathsf{p1}) = 24$, $w(\mathsf{p2}) = 42$, $w(\mathsf{p5}) = 90$, and $w(\mathsf{p10}) = 141$, contradicting the discussion at the end of [17, Example 2] claiming that a precedence $\mathsf{p10} > \mathsf{p5} > \mathsf{p2} > \mathsf{p1}$ is needed.

Without dependency pairs there is no real gain in speed when allowing rationals for SMT. This might be due to the fact that only two proofs (HM/t000 and SK90/2.46) make use of rational valued weights in the TRS category and three proofs when considering SRSs (Trafo/dup01, Trafo/dup11, and Trafo/dup16). But the difference becomes larger within the dependency pair setting. Suddenly 46 proofs for TRSs contain rational valued weights while the number does not increase for SRSs. As already mentioned earlier [53] proves that restricting to integer weights does not change the power of the order.

While running the experiments, sat and pbc produced different answers for the SRS Zantema/z13; pbc claimed KBO termination whereas sat answered "don't know". Chasing that discrepancy revealed a bug (Eén Niklas, personal conversation, 2007) in MiniSat+ (which has been corrected in the meantime).

Our experiments reveal that SMT suits an efficient and simple implementation best. However, for KBO linear arithmetic is sufficient which is not the case for other popular termination techniques like polynomial interpretations [58]. Currently SMT solvers do not support non-linear arithmetic at all or completely inappropriate. Thus until recently it seemed inevitable to use SAT as a backend [23] for efficient implementations dealing with polynomials. The approach from [8] allows to solve non-linear arithmetic constraints using SMT solvers only supporting linear arithmetic.

Comparing KBO with other direct approaches for proving termination such as LPO [45] or polynomial interpretations, the question arises how powerful KBO is. Despite the severe restriction of non-duplication, there are KBO terminating TRSs that cannot be oriented by LPO or polynomial interpretations. Taking derivational complexity as measure for the power of an order, KBO surpasses some other approaches. The *derivation length*, denoted $\mathrm{dl}_{\mathcal{R}}(n)$, computes the length of a longest possible derivation starting at a term of size n. Hofbauer and Lautemann [40] showed that KBO can prove TRSs $\mathcal{R}$ terminating for which $\mathrm{dl}_{\mathcal{R}}$ cannot be bounded by a primitive recursive function whereas polynomials are bounded from above by double exponential functions. Lepper [59] proved that the Ackermann function gives an upper bound on $\mathrm{dl}_{\mathcal{R}}$ if termination of $\mathcal{R}$ can be proved by KBO. Moser [66] extended this result to infinite signatures. Concerning LPO, Weiermann [86] showed that multiple recursive functions suffice for bounding derivational complexity.

To stress the significance of our contribution we state some details about the use of KBO in the termination prover T$_\text{T}$T$_2$. It used the SMT encoding (with rationals) in the November 2008 termination competition for both categories in which it participated. Here KBO had to compete with a number of other ter-

mination techniques in TTT2. For the category TRS Standard (SRS Standard) KBO was used in about 22% (18%) of the successful termination proofs which shows the applicability of the method.

2.7 Summary

In this chapter we revisited the Knuth-Bendix order (KBO) by presenting logic-based encodings of KBO—solvable by pure SAT, PB, and SMT—which can be implemented more efficiently and with considerably less effort than the dedicated methods described in [17, 53]. A method to compute upper bounds for weights makes the SAT and PB approaches complete. Furthermore we gave an alternative poly-time decidability result of the order to [53]. Comparisons with existing implementations on standard testbenches reveal enormous gains in efficiency. Especially the SMT approach gives rise to a very fast and user-friendly implementation since the method is parameter-free (no restriction of bits for weights).

Chapter 3

Increasing Interpretations

This chapter introduces a refinement of interpretation-based termination criteria for TRSs in the dependency pair setting. Traditional methods share the property that—in order to be successful—all rewrite rules considered must (weakly) decrease with respect to some measure. One novelty of the approach introduced in this chapter is that it allows an increase for some rules. Possible candidates for such rules are found by simultaneously searching for adequate polynomial interpretations while considering the information of the dependency graph. We prove that our method extends the termination proving power of linear interpretations. Furthermore, this generalization perfectly fits the dependency pair framework which is implemented in virtually every termination prover dealing with term rewrite systems. We present two DP processors for increasing interpretations. The novelty of the second one is that it can be used to eliminate single edges from the dependency graph. Two SAT encodings are given to implement the proposed DP processors and the implementations are evaluated on standard testbenches.

The chapter is organized as follows. In Section 3.1 the necessary preliminaries for the later sections are recalled. Section 3.2 motivates our approach by means of an example and already suggests that special care is needed to formulate a sound DP processor. Afterwards in Section 3.3 two sound and complete DP processors are introduced. Implementation details and optimizations are presented in Section 3.4. An assessment of our contribution can be found in Section 3.5 before ideas for future work are addressed in Section 3.6.

The results of this chapter originate from [92] which is a significant extension of [90].

3.1 Preliminaries

Next we fix notation on labeled graphs in Section 3.1.1 and introduce polynomial interpretations in Section 3.1.2.

3.1.1 Graphs

Let N be a finite set. A *graph* $\mathcal{G} = (N, E)$ is a pair such that $E \subseteq N \times N$. Elements of N (E) are called *nodes* (*edges*). A *labeled graph* is a pair $(\mathcal{G}, \ell)$ consisting of a graph $\mathcal{G} = (N, E)$ and a labeling function $\ell\colon N \to \mathbb{L}$ that assigns

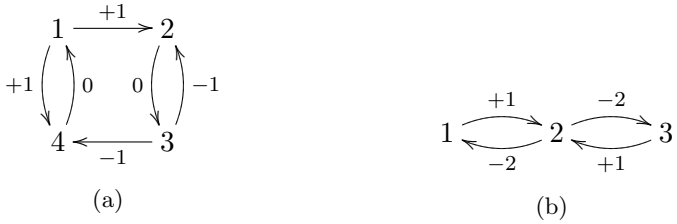


Figure 3.1: Labeled graphs

to every node a label.[1] A *path* from n_1 to n_m in a graph $\mathcal{G} = (N, E)$ is a finite sequence $[n_1, \ldots, n_m]$ of nodes such that $(n_i, n_{i+1}) \in E$ for all $1 \leqslant i < m$. In the sequel we only consider non-empty paths, i.e., $m > 1$. A path is called *elementary* if all its nodes are distinct. The *cost* of a path $[n_1, \ldots, n_{m-1}, n_m]$ is $\ell(n_1) + \cdots + \ell(n_{m-1})$ and its *length* is $m - 1$. The *distance* between two nodes a and b is the maximal cost of an elementary path from a to b and denoted by $\mathsf{d}(a, b)$ if such a path exists and undefined otherwise. A path $[n_1, \ldots, n_m]$ is called *cyclic* if $n_1 = n_m$. A *cycle* $[n_1, \ldots, n_m]$ is a cyclic path where $(n_i, n_{i+1}) \neq (n_j, n_{j+1})$ for all $1 \leqslant i < j < m$. A cycle $[n_1, \ldots, n_{m-1}, n_m]$ is called *elementary* if $n_1, \ldots, n_{m-1}$ are pairwise distinct. The definition of cost carries over naturally from paths to cycles. A cycle is called *decreasing* (*increasing*) if its cost is less (greater) than zero. Furthermore we define the *distance* $\mathsf{d}(n)$ for a single node n as the maximal cost of an elementary cycle starting in n if such a cycle exists. For aesthetic reasons, labels of nodes are associated to edges in graphical representations of graphs throughout the chapter, where edges (n, m) are labeled with $\ell(n)$.

Example 3.1. In the labeled graph of Figure 3.1(a), $p_1 = [1, 2, 3, 4, 1]$ is an example of a (non-elementary) path and also forms an elementary cycle. The (non-elementary) path $p_2 = [1, 4, 1, 4, 1]$ is no cycle since the edge $(1, 4)$ appears twice. We have $\text{length}(p_1) = 0$ and $\text{length}(p_2) = 2$. The distance of node 1 is 1 since it is the maximal cost of the elementary cycles $[1, 4, 1]$ and $[1, 2, 3, 4, 1]$. Note that in Figure 3.1(b) $[1, 2, 3, 2, 1]$ is a cycle but not an elementary one since node 2 is passed twice.

3.1.2 Polynomial Interpretations

Before recalling polynomial interpretations [58] over the natural numbers, the theoretical foundations why they may be used for termination proofs are presented.

For a signature $\mathcal{F}$ an $\mathcal{F}$-algebra $\mathcal{A}$ consists of a *carrier* A and a set of interpretations f_A for every $f \in \mathcal{F}$. Whenever $\mathcal{F}$ is irrelevant or clear from the context we call an $\mathcal{F}$-algebra simply algebra.

[1] In our setting we deal not only with concrete labels (e.g. $\mathbb{N}$, $\mathbb{Z}$, or $\mathbb{Q}$) but also with abstract labels (e.g. propositional formulas representing numbers in binary) that are closed under certain operations (addition, subtraction, maximum) and allow comparison. Further details are discussed in Section 3.4.

Definition 3.2. A non-empty $\mathcal{F}$-algebra $\mathcal{A}$ over the carrier A together with two relations $\geqslant$ and $>$ on A is called *weakly monotone*, if

- f_A is monotone in all its coordinates with respect to $\geqslant$, i.e., for all $a_1, \ldots, a_n, b \in A$ and $1 \leqslant i \leqslant n$:

$$a_i \geqslant b \text{ implies } f_A(a_1, \ldots, a_i, \ldots, a_n) \geqslant f_A(a_1, \ldots, b, \ldots, a_n),$$

- $>$ is well-founded, and
- $> \cdot \geqslant \subseteq >$ (compatibility) holds.

Let $\mathcal{A}$ be an algebra over some carrier A. An *assignment* α for $\mathcal{A}$ is a mapping from $\mathcal{V}$ to A. Interpretations are lifted from function symbols to terms—using assignments to fix the values of the variables—as usual. The induced mapping is denoted by $[\alpha]_{\mathcal{A}}(\cdot)$. For two terms s and t we define $s >_{\mathcal{A}} t$ if $[\alpha]_{\mathcal{A}}(s) > [\alpha]_{\mathcal{A}}(t)$ holds for all possible assignments α. The comparison $\geqslant_{\mathcal{A}}$ is similarly defined. Whenever α is irrelevant we abbreviate $[\alpha]_{\mathcal{A}}(s)$ to $[s]_{\mathcal{A}}$. Next we recall that weakly monotone algebras give rise to reduction pairs.

Theorem 3.3. *Let $(\mathcal{A}, \geqslant, >)$ be a weakly monotone algebra. Then $(\geqslant_{\mathcal{A}}, >_{\mathcal{A}})$ is a reduction pair.*

Proof. Immediate from [22, Theorem 2, part 2] which is a stronger result. □

Due to Theorems 1.7 and 3.3 we get the following corollary.

Corollary 3.4. *Let $(\mathcal{A}, \geqslant, >)$ be a weakly monotone algebra. The processor that maps a DP problem $(\mathcal{P}, \mathcal{R})$ to*

- $\{(\mathcal{P} \setminus >_{\mathcal{A}}, \mathcal{R})\}$ *if* $\mathcal{P} \subseteq \geqslant_{\mathcal{A}} \cup >_{\mathcal{A}}$ *and* $\mathcal{R} \subseteq \geqslant_{\mathcal{A}}$
- $\{(\mathcal{P}, \mathcal{R})\}$ *otherwise*

is sound and complete. □

In the sequel of this chapter we only regard a special class of weakly monotone algebras, namely polynomial interpretations [58]. Here every n-ary function symbol $f \in \mathcal{F}$ is mapped to a polynomial $f_{\mathbb{N}}$ over the carrier $\mathbb{N}$ in n indeterminates. Polynomial interpretations $\mathcal{I}$ together with $\geqslant_{\mathbb{N}}$ and $>_{\mathbb{N}}$ are algebras $(\mathcal{I}, \geqslant_{\mathbb{N}}, >_{\mathbb{N}})$. Hence polynomial interpretations which are monotone with respect to $\geqslant_{\mathbb{N}}$ over $\mathbb{N}$ yield reduction pairs. One important issue concerning automation is how polynomials are compared. For *non-linear* polynomials with coefficients ranging over the natural numbers this problem is known to be undecidable (Hilbert's 10^{th} problem). In typical implementations polynomials are ordered by absolute positiveness criteria [44]. In order to test whether $p > q$ holds for *linear* polynomials $p = c_0x_0 + \cdots + c_nx_n + c_{n+1}$ and $q = d_0x_0 + \cdots + d_nx_n + d_{n+1}$, a sufficient condition is $c_i \geqslant_{\mathbb{N}} d_i$ for all $0 \leqslant i \leqslant n$ and $c_{n+1} >_{\mathbb{N}} d_{n+1}$. The test $p \geqslant q$ is similar except for the constant case, i.e., $c_{n+1} \geqslant_{\mathbb{N}} d_{n+1}$.

Existing generalizations of polynomial interpretations allow different carriers, e.g., rational [64, 25] and real numbers [64] or integers [37, 39, 23, 24]. Other approaches go beyond pure polynomials and allow less restrictive kinds of interpretations including max [24]. Furthermore matrix [41, 22], quasi-periodic [99], and arctic [51] interpretations do also extend the termination proving power significantly. All these extensions share the property that the rewrite rules under consideration must weakly decrease and at least one rule has to decrease strictly. Our approach differs from these ones in the sense that we allow a possible increase for some rules (under the side condition that some other rules eliminate that increase). In order to detect possible candidates where the interpreted value might increase when applying a rule, the dependency pair method in combination with the dependency graph refinement is employed.

3.2 Towards Increasing Interpretations

This section demonstrates the limitations of polynomial interpretations and suggests an improvement by additionally considering the order of recursive calls encoded in the dependency graph.

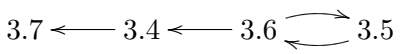
Example 3.5. Consider the TRS $\mathcal{R}$ consisting of the following three rules:

$$\mathsf{f}(0, x) \to \mathsf{f}(1, \mathsf{g}(x)) \tag{3.1}$$
$$\mathsf{f}(1, \mathsf{g}(\mathsf{g}(x))) \to \mathsf{f}(0, x) \tag{3.2}$$
$$\mathsf{g}(1) \to \mathsf{g}(0) \tag{3.3}$$

The dependency pairs

$$\mathsf{F}(0, x) \to \mathsf{G}(x) \tag{3.4}$$
$$\mathsf{F}(0, x) \to \mathsf{F}(1, \mathsf{g}(x)) \tag{3.5}$$
$$\mathsf{F}(1, \mathsf{g}(\mathsf{g}(x))) \to \mathsf{F}(0, x) \tag{3.6}$$
$$\mathsf{G}(1) \to \mathsf{G}(0) \tag{3.7}$$

admit the following dependency graph:



According to the DP processor of Theorem 1.5 it suffices to consider the DP problem $(\mathcal{P}, \mathcal{R})$ with $\mathcal{P} = \{3.5, 3.6\}$. To make further progress we have to find a reduction pair $(\gtrsim, >)$ such that all rules in $\mathcal{P} \cup \mathcal{R}$ decrease weakly and at least one rule in $\mathcal{P}$ decreases strictly. In the sequel we will show that the current DP problem $(\mathcal{P}, \mathcal{R})$ cannot be handled by reduction pairs based on traditional implementations of linear polynomial interpretations. To be able to address all possible polynomial interpretations, we consider our problem as an abstract constraint satisfaction problem. Consequently the coefficients for the polynomials are variables whose values are numbers. Similarly to [23] a term $\mathsf{F}(x, y)$ is transformed into an abstract linear polynomial $\mathsf{F}_0 x + \mathsf{F}_1 y + \mathsf{F}_2$. Doing

Table 3.1: Rules with increasing interpretations

$$\mathsf{f}(\mathsf{0},x) \to \mathsf{f}(\mathsf{1},\mathsf{g}(x)) \qquad 0 \geqslant 0 \tag{3.1}$$
$$\mathsf{f}(\mathsf{1},\mathsf{g}(\mathsf{g}(x))) \to \mathsf{f}(\mathsf{0},x) \qquad 0 \geqslant 0 \tag{3.2}$$
$$\mathsf{g}(\mathsf{1}) \to \mathsf{g}(\mathsf{0}) \qquad 1 \geqslant 1 \tag{3.3}$$
$$\mathsf{F}(\mathsf{0},x) \to \mathsf{F}(\mathsf{1},\mathsf{g}(x)) \qquad x \geqslant x+1 \tag{3.5}$$
$$\mathsf{F}(\mathsf{1},\mathsf{g}(\mathsf{g}(x))) \to \mathsf{F}(\mathsf{0},x) \qquad x+2 \geqslant x \tag{3.6}$$

so for the DP problem mentioned above results in the constraints

$$\mathsf{F}_0\mathsf{0}_0 + \mathsf{F}_1 x + \mathsf{F}_2 \geqslant \mathsf{F}_0\mathsf{1}_0 + \mathsf{F}_1(\mathsf{g}_0 x + \mathsf{g}_1) + \mathsf{F}_2$$
$$\mathsf{F}_0\mathsf{1}_0 + \mathsf{F}_1(\mathsf{g}_0(\mathsf{g}_0 x + \mathsf{g}_1) + \mathsf{g}_1) + \mathsf{F}_2 \geqslant \mathsf{F}_0\mathsf{0}_0 + \mathsf{F}_1 x + \mathsf{F}_2$$

where at least one inequality is strict. By simple mathematics the inequalities simplify to

$$\mathsf{F}_0\mathsf{0}_0 + \mathsf{F}_1 x \geqslant \mathsf{F}_0\mathsf{1}_0 + \mathsf{F}_1\mathsf{g}_0 x + \mathsf{F}_1\mathsf{g}_1. \tag{3.8}$$
$$\mathsf{F}_0\mathsf{1}_0 + \mathsf{F}_1\mathsf{g}_0\mathsf{g}_0 x + \mathsf{F}_1\mathsf{g}_0\mathsf{g}_1 + \mathsf{F}_1\mathsf{g}_1 \geqslant \mathsf{F}_0\mathsf{0}_0 + \mathsf{F}_1 x \tag{3.9}$$

From the fact that one of the above inequalities has to be strict it is obvious that $\mathsf{F}_1 > 0$. The constraints for x in (3.8) demand $\mathsf{g}_0 \leqslant 1$ and similarly (3.9) gives $\mathsf{g}_0 \geqslant 1$. Hence the constraint problem is equivalent to

$$\mathsf{F}_0\mathsf{0}_0 \geqslant \mathsf{F}_0\mathsf{1}_0 + \mathsf{F}_1\mathsf{g}_1 \tag{3.10}$$
$$\mathsf{F}_0\mathsf{1}_0 + \mathsf{F}_1\mathsf{g}_1 + \mathsf{F}_1\mathsf{g}_1 \geqslant \mathsf{F}_0\mathsf{0}_0 \tag{3.11}$$

which demands $\mathsf{g}_1 > 0$ to make one inequality strict. The (simplified) constraint for rule 3.3 amounts to

$$\mathsf{1}_0 \geqslant \mathsf{0}_0 \tag{3.12}$$

The proof is concluded by the contradictory sequence

$$\mathsf{F}_0\mathsf{0}_0 \geqslant \mathsf{F}_0\mathsf{1}_0 + \mathsf{F}_1\mathsf{g}_1 \geqslant \mathsf{F}_0\mathsf{0}_0 + \mathsf{F}_1\mathsf{g}_1 \tag{3.13}$$

where the first inequality derives from (3.10), the second one from (3.12), and the contradiction from the fact that $\mathsf{F}_1, \mathsf{g}_1 > 0$ which we learned earlier.

Although we just proved that there is no termination proof for the system above with linear polynomials, we will present a termination proof right now. Assume the weakly monotone interpretation

$$\mathsf{F}_{\mathbb{N}}(x,y) = x + y \qquad \mathsf{f}_{\mathbb{N}}(x,y) = 0 \qquad \mathsf{g}_{\mathbb{N}}(x) = x + 1 \qquad \mathsf{0}_{\mathbb{N}} = 0 \qquad \mathsf{1}_{\mathbb{N}} = 0$$

which orients almost all rules of interest correctly as can be seen in Table 3.1. Here, rule 3.5 is not correctly oriented. The idea to turn this interpretation

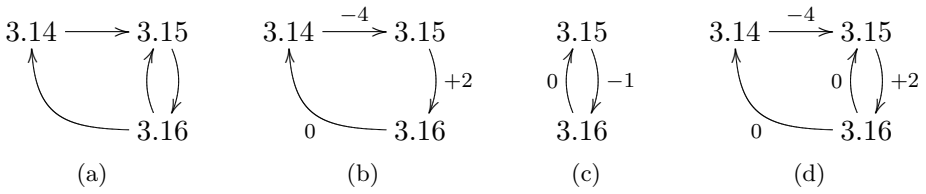
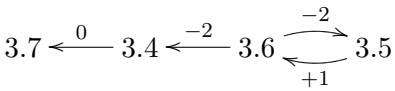


Figure 3.2: Different parts of (labeled) dependency graphs

into a valid termination proof is to combine the information of the dependency graph with the interpretation. From the (labeled) dependency graph



it becomes apparent that the two dependency pairs 3.5 and 3.6 are used alternately. The labels of the graph are computed as follows: From Table 3.1 one infers that an application of rule 3.6 *decreases* the interpreted value by the constant 2 (hence label -2) whereas rule 3.5 *increases* the value by the constant 1 (hence label $+1$). Consequently, after performing the cycle once the total value decreases by at least one. Therefore, the cycle cannot give rise to an infinite rewrite sequence (because the interpretation of every term is assumed to be non-negative).

Before moving to more challenging dependency graphs we stress that refinements such as usable rules do not allow linear polynomial interpretations to succeed on the previous example. The reason is that any interpretation that might remove one dependency pair must depend on F's second argument. But then the rule $\mathsf{g}(\mathsf{0}) \to \mathsf{g}(\mathsf{1})$ is usable and the same contradiction can be concluded. Similarly the argumentation is also not affected when changing to linear polynomials over the real coefficients.

3.2.1 From Cycles to SCCs

The above idea naturally extends from plain cycles to SCCs as described below. Consequently this allows to formulate a DP processor where DP problems of arbitrary shape can be handled. Nevertheless some care is needed when the dependency graph contains more complicated SCCs as the following example demonstrates. Consider the TRS $\mathcal{R}$ consisting of the five rules

$$\begin{aligned} \mathsf{f}(\mathsf{0},\mathsf{0},x,\mathsf{g}(\mathsf{g}(\mathsf{g}(\mathsf{g}(y))))) &\to \mathsf{f}(\mathsf{0},\mathsf{1},\mathsf{g}(\mathsf{g}(x)),y) & \mathsf{g}(\mathsf{0}) &\to \mathsf{g}(\mathsf{1}) \\ \mathsf{f}(\mathsf{0},\mathsf{1},\mathsf{g}(x),y) &\to \mathsf{f}(\mathsf{1},\mathsf{1},x,\mathsf{g}(\mathsf{g}(y))) & \mathsf{g}(x) &\to x \\ \mathsf{f}(\mathsf{1},\mathsf{1},x,y) &\to \mathsf{f}(\mathsf{0},x,x,y) \end{aligned}$$

which admits the single SCC in the dependency graph

$$\mathsf{F}(\mathsf{0},\mathsf{0},x,\mathsf{g}(\mathsf{g}(\mathsf{g}(\mathsf{g}(y))))) \to \mathsf{F}(\mathsf{0},\mathsf{1},\mathsf{g}(\mathsf{g}(x)),y) \tag{3.14}$$

$$\mathsf{F}(\mathsf{0},\mathsf{1},\mathsf{g}(x),y) \to \mathsf{F}(\mathsf{1},\mathsf{1},x,\mathsf{g}(\mathsf{g}(y))) \tag{3.15}$$

$$\mathsf{F}(\mathsf{1},\mathsf{1},x,y) \to \mathsf{F}(\mathsf{0},x,x,y). \tag{3.16}$$

(a) The labeled DG (b) The remaining DG

Figure 3.3: A hypothetically labeled DG

The corresponding part of the dependency graph depicted in Figure 3.2(a) contains the two cycles $[3.14, 3.15, 3.16, 3.14]$ and $[3.15, 3.16, 3.15]$. The first one is handled by the increasing interpretation

$$\mathsf{F}_{\mathbb{N}}(x, y, z, w) = w \qquad \mathsf{g}_{\mathbb{N}}(x) = x + 1 \qquad \mathsf{f}_{\mathbb{N}}(x, y, z, w) = \mathsf{0}_{\mathbb{N}} = \mathsf{1}_{\mathbb{N}} = 0.$$

For the second one we take the same interpretation as above but with a slightly altered value for $\mathsf{F}_{\mathbb{N}}$, namely $\mathsf{F}_{\mathbb{N}}(x, y, z, w) = z$. Hence for the elementary cycle $[3.14, 3.15, 3.16, 3.14]$ the interpreted value decreases by 2 in every loop. Similarly there is a decrease of 1 for the elementary cycle $[3.15, 3.16, 3.15]$. The two labeled graphs in Figures 3.2(b) and 3.2(c) describe the symbiosis of the interpretations and the elementary cycles. The only problem is that

$$\begin{aligned}
&\mathsf{f}(\mathsf{0}, \mathsf{0}, \mathsf{0}, \mathsf{g}(\mathsf{g}(\mathsf{g}(\mathsf{g}(\mathsf{g}(y)))))) \\
&\quad \to \mathsf{f}(\mathsf{0}, \mathsf{1}, \mathsf{g}(\mathsf{g}(\mathsf{0})), y) \to \mathsf{f}(\mathsf{1}, \mathsf{1}, \mathsf{g}(\mathsf{0}), \mathsf{g}(\mathsf{g}(y))) \\
&\quad \to \mathsf{f}(\mathsf{0}, \mathsf{g}(\mathsf{0}), \mathsf{g}(\mathsf{0}), \mathsf{g}(\mathsf{g}(y))) \to \mathsf{f}(\mathsf{0}, \mathsf{g}(\mathsf{1}), \mathsf{g}(\mathsf{0}), \mathsf{g}(\mathsf{g}(y))) \\
&\quad \to \mathsf{f}(\mathsf{0}, \mathsf{1}, \mathsf{g}(\mathsf{0}), \mathsf{g}(\mathsf{g}(y))) \to \mathsf{f}(\mathsf{1}, \mathsf{1}, \mathsf{0}, \mathsf{g}(\mathsf{g}(\mathsf{g}(\mathsf{g}(y))))) \\
&\quad \to \mathsf{f}(\mathsf{0}, \mathsf{0}, \mathsf{0}, \mathsf{g}(\mathsf{g}(\mathsf{g}(\mathsf{g}(y))))) \to \cdots
\end{aligned}$$

constitutes a non-terminating sequence in this TRS. What exactly went wrong can be seen when considering the whole SCC of the labeled dependency graph (using the first interpretation, cf. Figure 3.2(d)). In the conventional (non-increasing) setting it suffices to consider the two cycles separately. This is the case because a strict decrease in every single cycle ensures a strict decrease in larger cyclic paths by combining the partial proofs lexicographically. The example above shows that this is no longer true for increasing interpretations. The problematic non-terminating sequence corresponds to a path $[3.14, 3.15, 3.16, 3.15, 3.16, 3.14]$ where the interpreted value is increased in the elementary cycle $[3.15, 3.16, 3.15]$ and the cost of $[3.14, 3.15, 3.16, 3.15, 3.16, 3.14]$ is zero and there is no decrease. Considering (infinitely many!) possibly non-elementary cyclic paths is undoable. But if a graph contains no increasing cycles then the above scenario is not possible. To recognize dangerous cyclic paths, it suffices to compute the distance for every node. For the graph in Figure 3.2(d) we have $\mathsf{d}(3.14) = -2$, $\mathsf{d}(3.15) = 2$, and $\mathsf{d}(3.16) = 2$. Only if for every node the distance is smaller than or equal to zero we know that problematic paths as demonstrated above cannot occur. Furthermore we know that in such a case we can delete nodes with negative distance because on every possible cyclic path containing such a node the interpreted value decreases. If for the SCC under

consideration one had managed to find a weakly monotone interpretation with labeled dependency graph like the one in Figure 3.3(a) (which is of course impossible since the system at hand is not terminating) then deleting node 3.14 would have been possible since $\mathsf{d}(3.14) = -1$, $\mathsf{d}(3.15) = 0$, and $\mathsf{d}(3.16) = 0$. In such a situation one could proceed with the simpler graph depicted in Figure 3.3(b) with a possibly totally different interpretation.

3.3 Two DP Processors

The example in the preceding section suggests that DP problems that consist of more than just one cycle need special attention. Before formulating two DP processors we show how to label the dependency graph by a given interpretation $\mathcal{I}$. When considering a root rewrite step which applies a rule $s \to t$, the change of the interpreted value is $[t]_\mathcal{I} - [s]_\mathcal{I}$. The idea is to label every node by the constant part of that difference.

Definition 3.6. For a polynomial p we denote the *constant* (*non-constant*) part of p by $\mathsf{cp}(p)$ ($\mathsf{ncp}(p)$). For a term t and a polynomial interpretation $\mathcal{I}$ we write $\mathsf{ncp}_\mathcal{I}(t)$ as $\mathsf{ncp}([t]_\mathcal{I})$. This notation naturally extends to rules and TRSs, e.g., $\mathsf{ncp}_\mathcal{I}(s \to t) = \mathsf{ncp}_\mathcal{I}(s) \to \mathsf{ncp}_\mathcal{I}(t)$ and $\mathsf{ncp}_\mathcal{I}(\mathcal{R}) = \{\mathsf{ncp}_\mathcal{I}(s \to t) \mid s \to t \in \mathcal{R}\}$. The same notation is freely used for $\mathsf{cp}_\mathcal{I}$.

Definition 3.7. Let $\mathcal{I}$ be an interpretation and DG a dependency graph. The *labeled dependency graph* $\mathsf{DG}_\mathcal{I}$ is defined as (DG, ℓ) with labeling $\ell(s \to t) = \mathsf{cp}([t]_\mathcal{I} - [s]_\mathcal{I})$ for every node $s \to t$ in DG. By $\mathsf{d}_\mathcal{I}(n)$ we denote the distance of a node n in $\mathsf{DG}_\mathcal{I}$.

The above definition does not explicitly mention the set of labels $\mathbb{L}$. This is not necessary since closing the carrier A under subtraction yields $\mathbb{L}$. The next definition presents the first DP processor in the setting of increasing interpretations.

Definition 3.8. Let $(\mathcal{P}, \mathcal{R})$ be a DP problem, $\mathcal{G}$ the corresponding (part of the) dependency graph with nodes $\mathcal{P}$,[2] and $\mathcal{I}$ an interpretation. We define the DP processor $Proc_\mathcal{I}$ as follows: $Proc_\mathcal{I}(\mathcal{P}, \mathcal{R})$ returns

- $\{(\mathcal{P} \setminus \{p \in \mathcal{P} \mid \mathsf{d}_\mathcal{I}(p) < 0\}, \mathcal{R})\}$
  if $(\geqslant_\mathcal{I}, >_\mathcal{I})$ is a reduction pair, $\mathcal{R} \subseteq \geqslant_\mathcal{I}$, $\mathsf{ncp}_\mathcal{I}(\mathcal{P}) \subseteq \geqslant$, and $\mathsf{d}_\mathcal{I}(\mathcal{P}) \subseteq \mathbb{L}^{\leqslant 0}$
- $\{(\mathcal{P}, \mathcal{R})\}$ otherwise.

The above processor formulates that under the condition that no increasing cycles exist ($\mathsf{d}_\mathcal{I}(\mathcal{P}) \subseteq \mathbb{L}^{\leqslant 0}$), nodes that are only part of decreasing cycles ($\mathsf{d}_\mathcal{I}(p) < 0$) can be removed. Before we present the (first) main theorem we show that increasing interpretations subsume the conventional non-increasing ones. Together with the discussion in Section 3.2 this shows that increasing interpretations based on linear polynomials are really more powerful.

[2] Demanding that the nodes of $\mathcal{G}$ equal $\mathcal{P}$ is no restriction since one can always apply the dependency graph processor beforehand.

Lemma 3.9. *The DP processor of Definition 3.8 subsumes the one of Corollary 3.4 for linear polynomial interpretations.*

Proof. Assume that the processor of Corollary 3.4 applies. Hence there exists a reduction pair $(\succcurlyeq_{\mathcal{I}}, >_{\mathcal{I}})$ such that

$$\mathcal{R} \subseteq \succcurlyeq_{\mathcal{I}} \tag{3.17}$$

and

$$\mathcal{P} \subseteq \succcurlyeq_{\mathcal{I}} \cup >_{\mathcal{I}} \tag{3.18}$$

We have to show that under the assumption of (3.17) and (3.18) also

$$\mathcal{R} \subseteq \succcurlyeq_{\mathcal{I}} \tag{3.19}$$

$$\mathsf{ncp}_{\mathcal{I}}(\mathcal{P}) \subseteq \succcurlyeq \tag{3.20}$$

and

$$\mathsf{d}_{\mathcal{I}}(\mathcal{P}) \subseteq \mathbb{L}^{\leqslant 0} \tag{3.21}$$

holds. Constraint (3.17) is identical to (3.19). That (3.18) implies (3.20) is trivial for linear polynomials. For the other implication note that from (3.18) the property $\ell(\mathcal{P}) \subseteq \mathbb{L}^{\leqslant 0}$ follows which proves (3.21) (cf. Definition 3.7). Furthermore $s >_{\mathcal{I}} t$ implies $\mathsf{cp}_{\mathcal{I}}(t) > \mathsf{cp}_{\mathcal{I}}(s)$ which gives $\mathsf{d}_{\mathcal{I}}(s \to t) < 0$. This ensures that the processor of Definition 3.8 deletes the same nodes as the processor of Theorem 1.7. □

Theorem 3.10. *The DP processor $Proc_{\mathcal{I}}$ is sound and complete.*

Proof. To shorten notation we abbreviate $\mathcal{P} \setminus \{p \in \mathcal{P} \mid \mathsf{d}_{\mathcal{I}}(\mathcal{P}) < 0\}$ by $\mathcal{P}^0$ and $\mathcal{P} \setminus \mathcal{P}^0$ by $\mathcal{P}^{<0}$. First we show soundness. Suppose the DP problem $(\mathcal{P}^0, \mathcal{R})$ is finite. We have to show that $(\mathcal{P}, \mathcal{R})$ is finite. Suppose to the contrary that $(\mathcal{P}, \mathcal{R})$ is not finite. So there exists a minimal rewrite sequence

$$s_0 \to_{\mathcal{P}} t_0 \to^*_{\mathcal{R}} s_1 \to_{\mathcal{P}} t_1 \to^*_{\mathcal{R}} \cdots \tag{3.22}$$

We consider two cases:

- A rule $s \to t \in \mathcal{P}^{<0}$ is applied infinitely often in the sequence (3.22). Then one can extract a sequence

  $$s'_0 \to_{s \to t} t'_0 \to^*_{\mathcal{P} \cup \mathcal{R}} s'_1 \to_{s \to t} t'_1 \to^*_{\mathcal{P} \cup \mathcal{R}} \cdots$$

  Since $\succcurlyeq_{\mathcal{I}}$ is closed under contexts and substitutions, for all terms u, v, and all rules $l \to r \in \mathcal{R} \cup \mathcal{P}$ with $u \to_{l \to r} v$ we get $\mathsf{ncp}_{\mathcal{I}}(u) \succcurlyeq \mathsf{ncp}_{\mathcal{I}}(v)$. Because the infinite sequence was chosen such that the rule $s \to t$ is used infinitely often it is obvious that when starting from term s'_0 one must cycle in the dependency graph in order to reach s'_1. The fact that $\mathsf{d}_{\mathcal{I}}(s \to t) < 0$ together with $\mathsf{d}_{\mathcal{I}}(\mathcal{P}) \subseteq \mathbb{L}^{\leqslant 0}$ ensures that every cyclic path containing the

node $s \to t$ decreases the constant part of the interpretation strictly (note that $\mathsf{cp}_\mathcal{I}(\mathcal{R}) \subseteq \geqslant$ by definition). Hence, $\mathsf{cp}_\mathcal{I}(s'_0) > \mathsf{cp}_\mathcal{I}(s'_1)$. Repeating this argument gives rise to the sequence

$$\mathsf{cp}_\mathcal{I}(s'_0) > \mathsf{cp}_\mathcal{I}(s'_1) > \mathsf{cp}_\mathcal{I}(s'_2) > \mathsf{cp}_\mathcal{I}(s'_3) > \cdots$$

which contradicts the well-foundedness of $>$. Consequently there cannot be an infinite sequence (3.22) contradicting the assumption that $(\mathcal{P}, \mathcal{R})$ is not finite.

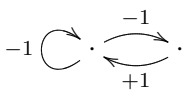
- No rule $s \to t \in \mathcal{P}^{<0}$ is applied infinitely often in (3.22). Then there exists a tail of (3.22) such that all $\mathcal{P}$-steps are from $\mathcal{P}^0$. By assumption the sequence (3.22) is minimal. Hence the tail is minimal as well and thus the DP problem $(\mathcal{P}^0, \mathcal{R})$ is not finite.

Completeness says that if $(\mathcal{P}^0, \mathcal{R})$ is not finite then $(\mathcal{P}, \mathcal{R})$ is not finite. This is trivial since any minimal $s_0 \to_{\mathcal{P}^0} t_0 \to^*_\mathcal{R} s_1 \to_{\mathcal{P}^0} t_1 \to^*_\mathcal{R} \cdots$ sequence is at the same time a minimal $s_0 \to_\mathcal{P} t_0 \to^*_\mathcal{R} s_1 \to_\mathcal{P} t_1 \to^*_\mathcal{R} \cdots$ sequence (because clearly $\mathcal{P}^0 \subseteq \mathcal{P}$). □

Next we show that increasing interpretations can be used to delete single edges in the dependency graph, resulting in a finer DP processor than the one of Definition 3.8. Similar as in [79] the dependency graph is added to DP problems—resulting in the notion of *extended* DP problems. An extended DP problem is a triple $(\mathcal{P}, \mathcal{R}, \mathcal{G})$. So in addition to ordinary DP problems now also some processing on the third component (the dependency graph) is possible. Ordinary DP processors remain sound if this third component is added. Furthermore the initial DP problem is now set to $(\mathsf{DP}(\mathcal{R}), \mathcal{R}, \mathsf{DG}(\mathcal{P}, \mathcal{R}))$ and for an infinite minimal sequence additionally it is demanded that consecutive $\mathcal{P}$ steps correspond to an edge in $\mathcal{G}$. It makes sense to redefine the processor of Theorem 1.5 such that for the (extended) DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ it returns the set of extended DP problems $\{(\mathcal{P}_i, \mathcal{R}, \mathcal{G}_i)\}$ such that $\mathcal{P}_i$ are the SCCs of $\mathcal{G}$ and $\mathcal{G}_i$ is the restriction of $\mathcal{G}$ to $\mathcal{P}_i$.

The following example motivates why deleting edges can be advantageous.

Example 3.11. In the labeled graph



all preconditions for applying the processor $Proc_\mathcal{I}$ are satisfied. But it cannot make progress since for all nodes n the property $\mathsf{d}(n) = 0$ holds. But clearly there cannot be an infinite run that uses the leftmost edge infinitely often.

Next the improved processor is presented that allows removing edges from the dependency graph. For this processor (and the discussion in Section 3.4.2) labels are associated to edges instead of nodes. A node-labeling ℓ_n is transformed into an edge-labeling ℓ_e as follows: $\ell_e((a, b)) = \ell_n(a)$ for every edge (a, b). Concepts as cost, distance, etc. carry over naturally from node-labeled to edge-labeled graphs. For reasons of readability we write $\ell_e(a, b)$ instead of $\ell_e((a, b))$ in the sequel.

Definition 3.12. Let $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ be an extended DP problem with $\mathcal{G} = (\mathcal{P}, \mathcal{E})$, $\mathcal{I}$ an interpretation, and $\mathcal{G}_\mathcal{I} = (\mathcal{G}, \ell)$ the labeled variant of $\mathcal{G}$. We define the DP processor $Proc'_\mathcal{I}$ as follows: $Proc'_\mathcal{I}(\mathcal{P}, \mathcal{R}, \mathcal{G})$ returns

- $\{(\mathcal{P}, \mathcal{R}, (\mathcal{P}, \mathcal{E} \setminus \{(a,b) \in \mathcal{E} \mid \ell_\mathcal{I}(a,b) + \mathsf{d}_\mathcal{I}(b,a) < 0\}))\}$
  if $(\geqslant_\mathcal{I}, >_\mathcal{I})$ is a reduction pair, $\mathcal{R} \subseteq \geqslant_\mathcal{I}$, $\mathsf{ncp}_\mathcal{I}(\mathcal{P}) \subseteq \geqslant$, and $\mathsf{d}_\mathcal{I}(\mathcal{P}) \subseteq \mathbb{L}^{\leqslant 0}$
- $\{(\mathcal{P}, \mathcal{R}, \mathcal{G})\}$ otherwise.

The idea of the processor above is that edges that appear on decreasing cycles only ($\ell(a,b) + \mathsf{d}(b,a) < 0$) do not give rise to infinite reductions.

Theorem 3.13. *The processor $Proc'_\mathcal{I}$ is sound and complete.*

Proof. The proof is quite similar to the one of Theorem 3.10. For soundness we assume a minimal infinite sequence that applies an edge $(s \to t, u \to v)$ infinitely often. This means that the minimal sequence applies rule $u \to v$ after $s \to t$ infinitely often (with possible $\mathcal{R}$-steps but without any $\mathcal{P}$-steps in between), i.e.,

$$\begin{aligned} s_0 &\to_{s\to t} t_0 \to^*_\mathcal{R} s_1 \to_{u\to v} t_1 \to^*_{\mathcal{P}\cup\mathcal{R}} s_2 \\ &\to_{s\to t} t_2 \to^*_\mathcal{R} s_3 \to_{u\to v} t_3 \to^*_{\mathcal{P}\cup\mathcal{R}} s_4 \\ &\to_{s\to t} \cdots \end{aligned}$$

But then the sequence

$$\mathsf{cp}_\mathcal{I}(s_0) > \mathsf{cp}_\mathcal{I}(s_2) > \mathsf{cp}_\mathcal{I}(s_4) > \cdots$$

gives the desired contradiction. This sequence exists because of the condition $\mathsf{d}_\mathcal{I}(\mathcal{P}) \subseteq \mathbb{L}^{\leqslant 0}$ and the assumption that the edge $(s \to t, u \to v)$ appears on decreasing cycles only (since $\ell_\mathcal{I}(s \to t, u \to v) + \mathsf{d}_\mathcal{I}(u \to v, s \to t) < 0$). Completeness is trivial. □

3.4 Implementation

Almost all fast implementations of polynomial interpretations are based on a transformation to a SAT problem. The major drawback is that one has to work with abstract encodings all the time. Hence when labeling the dependency graph one does not have concrete numbers at hand but some arithmetic constraints which abstractly encode the range of all possible values. Thus existing algorithms from graph theory cannot be employed because they require concrete numbers. Since encoding polynomials in SAT has already been described in detail [23], in this thesis we refrain from repeating all implementation issues. The only part of the encoding which is discussed here deals with graphs that are labeled by abstract numbers. This includes computing the distance $\mathsf{d}(n)$ for a node n and expressing if an edge (a,b) occurs on decreasing cycles only. Two methods to tackle this problem are discussed in the sequel. Below is the first approach. An alternative is presented in Section 3.4.2.

3.4.1 Computing the Distance of a Node

The first idea is to compute the distance of a node by means of a transitivity closure. The numerical variable R_{abi} is $-\infty$ if node b is not reachable in at most 2^i steps from node a and otherwise this variable keeps the (currently known) distance from a to b. It is obvious that in a graph (N, E) an elementary cycle contains at most $|N|$ edges and hence for $k' \geqslant k := \lceil \log_2(|N|) \rceil$ one has reached kind of a fixed point, i.e., $R_{abk'} = -\infty$ if and only if $R_{abk} = -\infty$ for all $a, b \in N$.

More precisely, the variables R_{ab0} reflect the edges of the graph and hence b is reachable from a with a cost of $\ell(a)$ if $(a, b) \in E$ and it is unreachable if $(a, b) \notin E$. Thus we initialize these variables as follows:

$$R_{ab0} = \begin{cases} \ell(a) & \text{if } (a,b) \in E, \\ -\infty & \text{otherwise.} \end{cases}$$

Since R_{abi} might be $-\infty$, the operations addition and maximum are extended naturally, i.e., $n + -\infty = -\infty + n = -\infty$ and $\max(n, -\infty) = \max(-\infty, n) = n$ for all $n \in \mathbb{L} \cup \{-\infty\}$. For $0 \leqslant i < k$ we define

$$R_{ab(i+1)} = \max\big\{R_{abi}, \max_{m \in N}\{R_{ami} + R_{mbi}\}\big\}.$$

If one first forgets about max then the above formula expresses that b is reachable from a in at most 2^{i+1} steps with a cost of $R_{ab(i+1)}$ if it is already reachable within 2^i steps with that cost or there is a mid-point[3] m and the cost from a to m and the one from m to b just sum up. Taking the maximum of all possible costs ensures that we consider a worst case scenario, i.e., a path of maximal cost. In the end we want to test if $R_{nnk} \leqslant 0$ for all $n \in N$. Note that it might happen that the value R_{nnk} does not emerge from an elementary cycle (because maybe some cyclic path is considered). Nevertheless the idea remains sound because of the condition that for all nodes $R_{nnk} \leqslant 0$ there are no increasing cycles. In such a case the problem sketched here does not occur. For a demonstration consider the following example.

Example 3.14. In the labeled graph from Example 3.1(a) on page 38 we compute $k = \lceil \log_2(4) \rceil = 2$ and

$$\begin{array}{llll} \mathsf{d}(1) = 1 & \mathsf{d}(2) = 0 & \mathsf{d}(3) = 0 & \mathsf{d}(4) = 1 \\ R_{112} = 2 & R_{222} = 0 & R_{332} = 0 & R_{442} = 2. \end{array}$$

The reason for the different values is that R_{112} does not correspond to an elementary cycle; we have $\mathsf{d}(1) = 1$ (see Example 3.1) but $R_{112} = 2$ since it derives from the cyclic path $[1, 4, 1, 4, 1]$. A similar argument explains the discrepancy of $\mathsf{d}(4)$ and R_{442}.

But nevertheless one can formulate the following two lemmata which allow to implement the DP processor of Definition 3.8.

[3] Fortunately Zeno of Elea was wrong and this approach constitutes a valid method for computing reachability.

Lemma 3.15. *Let $((N, E), \ell)$ be a labeled graph and $k = \lceil \log_2(|N|) \rceil$. If for all nodes $n \in N$ the property $R_{nnk} \leqslant 0$ holds then for all nodes $n \in N$ the property $\mathsf{d}(n) \leqslant 0$ holds.*

Proof. Assume for a contradiction that there is a node $n \in N$ with $\mathsf{d}(n) > 0$. Hence there exists an elementary cycle that is increasing and contains n. But then clearly this cycle gives $R_{nnk} > 0$. □

Lemma 3.16. *Let $((N, E), \ell)$ be a labeled graph, $k = \lceil \log_2(|N|) \rceil$, and for all $n \in N$ $\mathsf{d}(n) \leqslant 0$. Then $\mathsf{d}(a, b) = R_{abk}$ for all distinct nodes $a, b \in N$ and $\mathsf{d}(a) = R_{aak}$ for all $a \in N$.*

Proof. We show the first equality, the second one is proved analogously. The distance from a to b is the maximal cost of an elementary path. Clearly by construction $R_{abk} \geqslant \mathsf{d}(a, b)$ since every elementary path from a to b is covered by an R_{abk} variable. By assumption there are no increasing cycles and hence non-elementary paths from a to b cannot have a larger cost than an elementary one. Hence R_{abk} represents the maximal cost of an elementary path from a to b. □

Both DP processors from Section 3.3 can be implemented with the help of the R_{abi} variables. For the processor from Definition 3.8 the formula (in the sequel referred to as direct_n)

$$\bigwedge_{n \in N} \big(R_{nnk} \leqslant 0\big) \ \wedge \bigvee_{n \in N} \big(R_{nnk} < 0\big)$$

is employed whereas Definition 3.12 amounts to the formula (direct_e)

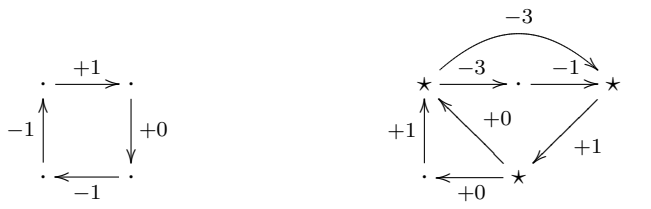
$$\bigwedge_{n \in N} \big(R_{nnk} \leqslant 0\big) \ \wedge \bigvee_{(a,b) \in E} \big(\ell(a, b) + R_{bak} < 0\big).$$

Then from a satisfying assignment the nodes with negative distance or edges that are just part of decreasing cycles can easily be determined and removed. Clearly satisfiability of the first formula implies satisfiability of the second formula. The reason why both are presented is that the first one produces a smaller encoding. That this can be advantageous is demonstrated by experimental data in Section 3.5.

Special Algorithms

The encoding for computing maximal paths as arithmetic constraints based on the R_{abi} variables from above has complexity $\mathcal{O}(n^2 \log(n))$ where n is the number of nodes in the underlying SCC of the labeled graph. To get a faster implementation we can optimize the encoding for SCCs that have a special shape:

(a) Simple SCCs: An SCC is called *simple* if it contains exactly one cycle, i.e., omitting any edge would destroy the property of being an SCC. An example of this shape is depicted in Figure 3.4(a). Linear time suffices to



(a) A simple SCC (b) An almost simple SCC

Figure 3.4: Two special shapes of SCCs

decide if a given SCC $\mathcal{S}$ is simple (the number of edges equals the number of nodes). In such a case the encoding specializes to

$$\sum_{n \in \mathcal{S}} \big(\ell(n) < 0\big)$$

which expresses that the constant part of the interpretation decreases when cycling. The encoding is linear in the size of the nodes.

(b) Almost simple SCCs: An SCC is called *almost simple* if there exists a node n (called *selected node*) such that after deleting *all* outgoing edges of n there is no non-empty sub-SCC left. Here we will exploit the fact that in every elementary cycle within this SCC we pass the node n. The nodes indicated with $\star$ in Figure 3.4(b) satisfy this property. In the encoding we demand that $-(\ell(n_1) + \cdots + \ell(n_p)) > \ell(m_1) + \cdots + \ell(m_q)$ holds where $n_1, \ldots, n_p$ are the selected nodes and $m_1, \ldots, m_q$ are the (non-selected) nodes in the SCC that have a positive label. The underlying idea is that the selected nodes n_i decrease the interpretation more than all other nodes together might increase it. In Figure 3.4(b) the inequality amounts to $-(-3+0+1) > 1$ which is satisfiable since $2 > 1$. The encoding again is linear in the size of the nodes. Furthermore it specializes exactly to the one in (a) for simple SCCs.

Case (a) is exact whereas (b) is an approximation, i.e., there are graphs where a node with negative distance is undetected. E.g., the graph in Figure 3.1(b) yields $-(-2) > 1 + 1$ which is not satisfiable although $\mathsf{d}(2) = -1$.

3.4.2 Compressing Graphs

This section presents a solution to implement (a strong heuristic for) the DP processor of Definition 3.12 by processing the graph. In an iterative manner nodes are removed from the graph one by one. While removing nodes, information about the cycles in the graph is obtained which allows to formulate a constraint such that satisfiability of this constraint implies the existence of a decreasing cycle. At the end of this section we discuss how this approach allows to implement the improved DP processor from the previous section.

For a simpler presentation of the algorithm, labels of non-existing edges are assumed to be $-\infty$. The intuition behind Definition 3.17 is as follows. In step 1

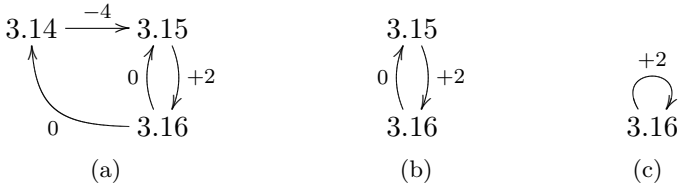


Figure 3.5: Compressing labeled graphs

cycles of length one are removed from the graph. Then in steps 2 and 3 a graph (N', E') is built that contains all edges of the graph from step 1, except that the chosen node n is bypassed. The bypass edges are exactly the ones in $i(n) \times o(n)$ where $i(n) = \{m \in N \mid (m, n) \in E\}$ and $o(n) = \{p \in N \mid (n, p) \in E\}$. Bypassing nodes might produce a multi-graph. Reasoning about multi-graphs is prevented by altering the labeling function ℓ which chooses the maximal edge between two nodes in step 4. Note that if $a \notin i(n)$ or $b \notin o(n)$ then $\ell'(a, b) = \ell''(a, b)$.

Definition 3.17 (Compression Algorithm)**.** Given a labeled graph $((N, E), \ell)$ the procedure works as follows. Set $C = \varnothing$.

1. Set $C = C \cup \{\ell(a, a) \mid \ell(a, a) \neq -\infty\}$, $E'' = E \setminus \{(a, a) \mid a \in N\}$, and
$$\ell''(a, b) = \begin{cases} \ell(a, b) & \text{if } a \neq b, \\ -\infty & \text{otherwise.} \end{cases}$$
2. If $|N| = 1$ then return C, otherwise choose a node $n \in N$.
3. Set $N' = N \setminus \{n\}$ and $E' = (E'' \cap N' \times N') \cup i(n) \times o(n)$.
4. Set $\ell'(a, b) = \max\{\ell''(a, b), \ell''(a, n) + \ell''(n, b)\}$.
5. Repeat step 1 with $((N', E'), \ell')$.

The next example performs the compression algorithm on a concrete graph.

Example 3.18. Consider the graph depicted in Figure 3.5(a). Step 1 is easy since there are no cycles of length one. In step 2 we select node 3.14. Clearly $i(3.14) = \{3.16\}$ and $o(3.14) = \{3.15\}$. Hence in step 3 we obtain $N' = \{3.15, 3.16\}$ and $E' = \{(3.15, 3.16), (3.16, 3.15)\}$. In step 4 the labeling ℓ'' looks like $\ell''(3.15, 3.16) = +2$ and $\ell''(3.16, 3.15) = \max\{0, 0 - 4\} = 0$ and all other labels are $-\infty$. The result of the algorithm after one iteration is shown in Figure 3.5(b). In the second iteration there are no changes in step 1. Now choose $n = 3.15$. This results in a graph with only one remaining node 3.16 and $\ell''(3.16, 3.16) = +2$, shown in Figure 3.5(c). In the next iteration the algorithm sets $C = \{+2\}$, $\ell''(3.16, 3.16) = -\infty$, and terminates in step 2.

The property that all cycles are non-increasing is fulfilled if for all values $c \in C$ we have $c \leqslant 0$ which is not the case in the example above. Here Definition 3.17 was applied to a labeled graph with concrete labels. In the setting of increasing

interpretations the algorithm is applied to a labeled dependency graph (with abstract labels) and thus returns a set of abstract numbers. To demand that there is no increase in any elementary cycle the expression $\bigwedge_{c \in C}(c \leqslant 0)$ (correctness) is employed. Furthermore progress (that there is a decreasing cycle) is achieved by $\bigvee_{c \in C}(c < 0)$. In contrast to the approach with the variables R_{abi} one cannot directly determine from a satisfying assignment which nodes (if any) have negative distance and thus can be removed. But from a satisfying assignment the concrete labels can be inferred. Afterwards maximal distances can easily be computed. Additionally edges (a, b) which are just contained in decreasing cycles ($\ell(a, b) + \mathsf{d}(b, a) < 0$) can be searched and removed instantaneously. This allows to implement the DP processor from Definition 3.12. But first we prove the correctness of the approach.

Lemma 3.19. *One iteration of the algorithm in Definition 3.17 fulfills the property that there exists a node $x \in N$ with $\mathsf{d}(x) > 0$ if and only if there exists a node $x' \in N'$ with $\mathsf{d}(x') > 0$ or there exists a $c \in C$ with $c > 0$.*

Proof. Assume there is a node $x \in N$ with $\mathsf{d}(x) > 0$. We consider two cases:

- There is a node y with $\ell(y, y) > 0$. Then in step 1 $\ell(y, y) \in C$ and thus the claim holds.
- There is no y with $\ell(y, y) > 0$. We again consider two cases.
  - The chosen node n is on a cycle involving x with positive distance. By the assumptions such a cycle must contain at least two different nodes and since x is existentially quantified we can choose it such that $x \neq n$. Hence we may denote the cycle by $[x, \ldots, a, n, b, \ldots, x]$. But then also $\mathsf{d}(x) > 0$ for the cycle $[x, \ldots, a, b, \ldots, x]$ in N' because by construction $\ell'(a, b) \geqslant \ell''(a, n) + \ell''(n, b)$. Hence the result holds by taking $x' = x$.
  - If the chosen node n is not on a cycle involving x then we take $x' = x$ and the result trivially holds.

For the other direction again two cases are considered:

- First assume there is a node $x' \in N'$ such that $\mathsf{d}(x') > 0$. Without loss of generality we can assume that a bypass edge (a, b) is contained in the cycle $[x', \ldots, a, b, \ldots, x']$. By construction $\ell'(a, b) = \ell''(a, b)$ or $\ell'(a, b) = \ell''(a, n) + \ell''(n, b)$. Choosing the appropriate edge(s) gives a cycle in the original graph with $\mathsf{d}(x) > 0$ where $x = x'$.
- The other case is if there is a $c \in C$ with $c > 0$. Then in step 1 there must be a node $x \in N$ with $\ell(x, x) > 0$ ensuring $\mathsf{d}(x) > 0$.

This concludes the proof. □

The next corollary states that the compression algorithm introduced in Definition 3.17 is sound and thus can be used to implement the DP processor from Definition 3.12.

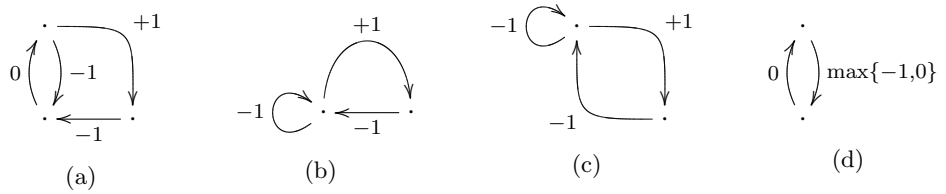


Figure 3.6: Problems of the compression algorithm

Corollary 3.20. *Let $((N, E), \ell)$ be a labeled graph. The following properties are equivalent:*

- *The distance $\mathsf{d}(x) \leqslant 0$ for all $x \in N$.*
- *The conjunction $c \leqslant 0$ for $c \in C$ is satisfiable where C is the output of the algorithm in Definition 3.17.* □

This means that whenever the conjunction of $\bigwedge_{c \in C}(c \leqslant 0)$ is satisfiable then $\mathsf{d}(n) \leqslant 0$ for all $n \in N$ (which is demanded by $Proc'_{\mathcal{I}}$). However in order to ensure progress of the processor there must exist a $c \in C$ with $c < 0$. In the sequel we abbreviate the conjunction of $\bigwedge_{c \in C}(c \leqslant 0)$ and $\bigvee_{c \in C}(c < 0)$ by $\mathrm{C}^{\wedge}_{\vee}$. Note that one cannot conclude from the satisfiability of $\mathrm{C}^{\wedge}_{\vee}$ the existence of a node n with $\mathsf{d}(n) < 0$. Example 3.11 shows such a case. But satisfiability of $\mathrm{C}^{\wedge}_{\vee}$ ensures a decreasing cycle. Thus the removal of edges as in the processor of Definition 3.12 applies.

To further demonstrate the compression algorithm the labeled graph from Figure 3.6(a) is considered. Clearly the leftmost edge labeled -1 cannot be taken indefinitely and could be removed since it is part of decreasing cycles only. But depending on the selection of the nodes in the compression algorithm, the formula $\mathrm{C}^{\wedge}_{\vee}$ may be satisfiable or not. The reason is that multi-edges are removed by maximizing over the labels and consequently decreasing cycles can be overlooked. Figures 3.6(b), 3.6(c), and 3.6(d) show the result of the algorithm after one step if the left-top, left-bottom, and right-bottom node is chosen to be eliminated. Hence our implementation provides just a heuristic for the processor $Proc'_{\mathcal{I}}$. However if there exists a node with negative distance, then maximizing does not destroy satisfiability of $\mathrm{C}^{\wedge}_{\vee}$ (since all elementary cycles containing this node are decreasing).

3.5 Assessment

In this chapter we showed that increasing interpretations are strictly more powerful than standard (non-increasing) interpretations over linear polynomials. Clearly for DP problems $(\mathcal{P}, \mathcal{R})$ where $\mathcal{P}$ is a singleton set they are of equal power.

The reason why the TRS of Example 3.5 cannot be proved terminating by means of linear polynomials is that we cannot differentiate constant $\mathsf{0}$ from $\mathsf{1}$ by the interpretation. Hence it is not so astonishing that the problematic SCC

can be handled by matrix interpretations [22] of dimension two. Actually all of the tools (dedicated to proving termination) participating in the TRS category of the 2007 and 2008 editions of the international termination competition can handle this system. All the proofs rely on matrix interpretations with dimension two. As a pre-processing step AProVE [30] and $\mathsf{T_TT_2}$ use dependency pair analysis whereas Jambox [21] performs a reduction of right-hand sides [97].

It is an easy exercise to construct (larger) TRSs than Example 3.5 such that all tools of the termination competition fail. To disallow Jambox the rewriting of right-hand sides we introduce overlaps. To knock-out the matrix method just increasing the size of the system suffices. Since $\mathsf{T_TT_2}$ can still prove these examples by bounds [28, 54] we ensure the TRS to be not left-linear which makes increasing interpretations the only successful method.

Example 3.21. The TRS below resembles a binary counter. The rules behave in a way such that the counter increments by one (adding a function symbol i in f's fourth argument) seven times in a row before the value is decremented by eight.

$$\begin{aligned}
\mathsf{f}(0,0,0,x) &\to \mathsf{f}(0,0,1,\mathsf{i}(x)) & \mathsf{f}(0,0,1,x) &\to \mathsf{f}(0,1,0,\mathsf{i}(x))\\
\mathsf{f}(0,1,0,x) &\to \mathsf{f}(0,1,1,\mathsf{i}(x)) & \mathsf{f}(0,1,1,x) &\to \mathsf{f}(1,0,0,\mathsf{i}(x))\\
\mathsf{f}(1,0,0,x) &\to \mathsf{f}(1,0,1,\mathsf{i}(x)) & \mathsf{f}(1,0,1,x) &\to \mathsf{f}(1,1,0,\mathsf{i}(x))\\
\mathsf{f}(1,1,0,x) &\to \mathsf{f}(1,1,1,\mathsf{i}(x)) & \mathsf{f}(y,y,y,\mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{i}(\mathsf{i}(x)))))))) &\to \mathsf{f}(0,0,0,x)\\
\mathsf{i}(\mathsf{i}(0)) &\to 1 & \mathsf{i}(\mathsf{i}(1)) &\to \mathsf{i}(\mathsf{i}(0))
\end{aligned}$$

None of the current termination tools succeeds in proving termination within a 60 seconds time limit. Increasing interpretations produce a successful—and very intuitive—proof for the challenging SCC. It considers the changes of F's fourth argument. The general approaches described in Section 3.4.1, the specialization (b) from Section 3.4.1, and the compression approach from Section 3.4.2 yield the increasing interpretation

$$\mathsf{F}_{\mathbb{N}}(x,y,z,w) = w \qquad \mathsf{i}_{\mathbb{N}}(x) = x+1 \qquad \mathsf{f}_{\mathbb{N}}(x,y,z,w) = 0_{\mathbb{N}} = 1_{\mathbb{N}} = 0$$

which ensures that all nodes have a negative distance and hence the whole problematic SCC can be removed. The only difference between the four approaches is that it takes the direct methods more than a second whereas the other approaches succeed within a fraction of a second.

We compare the different approaches from this chapter in Table 3.2. The exact strategies used for generating the table with $\mathsf{T_TT_2}$ are listed in Section A.4. In the table direct refers to the direct approach introduced in Section 3.4.1, where the suffix _n (_e) indicates if the formula to remove nodes (edges) is employed. Rows a and b indicate the usage of the optimizations (a) and (b) from Section 3.4.1 and compress the compression algorithm presented in Definition 3.17. The column labeled yes indicates the number of TRSs that could be proved terminating. The second column presents the total time in seconds used by the specific method. If the algorithm could not determine a result within 60

Table 3.2: Experimental results

(a) 1391 TRSs

method	yes	time	t/o
direct_n	478	18106	255
direct_e	477	18808	261
a	203	220	0
b	223	1726	12
compress	532	9447	123

(b) 732 SRSs

method	yes	time	t/o
direct_n	37	5321	54
direct_e	36	6046	60
a	10	14	0
b	13	129	0
compress	47	651	4

seconds, the computation was aborted (column t/o). In Table 3.2 linear polynomials with coefficients from $\{0, \ldots, 3\}$ have been employed. Intermediate results are restricted to numbers less than 7. Summing up, the compression algorithm performs better than the direct approach in both measurements: time and termination proving power (due to the time limit). Especially for SRSs the compression algorithm is much faster as can be inferred from Table 3.2(b). The tables also show that the optimizations from Section 3.4.1 allow a fast implementation for special cases that appear surprisingly frequent. We conclude this section with a remark on the compression algorithm. Choosing the node n to bypass drastically influences performance. Our heuristic prefers nodes n where $i(n) \times o(n)$ is small. Consequently less max operations are needed to combine labels of multiple edges between two nodes to prevent reasoning about multi-graphs.

3.6 Related and Future Work

The theory of increasing interpretations as described above directly applies to other reduction pairs based on interpretations. Both DP processors (Definitions 3.8 and 3.12) are formulated (and proved) for arbitrary reduction pairs based on interpretations. The only special requirement—needed for labeling the dependency graph—is that the constant part of the interpretation of a dependency pair is a number. Hence increasing interpretations also admit polynomial interpretations with rational [64, 25] and real coefficients [64] as well as negative constants [39]. Furthermore they can also be combined with the matrix method [22] which is introduced in the next chapter since there the constant part of the interpretation of a dependency pair amounts to a number (cf. Example 4.6). Consequently the dependency graph is labeled in exactly the same fashion. The recent technique from [24] to allow the maximum operation can also be used in combination with the method proposed here. The reason is that the (more complex) max-polynomials only occur in intermediate steps; when labeling the dependency graph, all occurrences of max have already been removed.

Generalizing the approach in such a way that not only the constant part of the interpretation is used as additional information in the dependency graph but also the non-constant part, is highly desirable. We anticipate that this

would make the approach significantly more powerful. The only drawback is that probably this generalization applies to a very restricted class of TRSs only. To get a feeling for the problems that arise consider the non-terminating system consisting of the two rules

$$\mathsf{f}(\mathsf{s}(x)) \to \mathsf{g}(\mathsf{s}(x)) \qquad\qquad \mathsf{g}(x) \to \mathsf{f}(x)$$

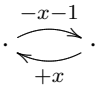
which admits the dependency pairs

$$\mathsf{F}(\mathsf{s}(x)) \to \mathsf{G}(\mathsf{s}(x)) \qquad\qquad \mathsf{G}(x) \to \mathsf{F}(x).$$

The increasing interpretation with

$$\mathsf{F}_{\mathbb{N}}(x) = 2x \qquad \mathsf{G}_{\mathbb{N}}(x) = x \qquad \mathsf{f}_{\mathbb{N}}(x) = 0 \qquad \mathsf{s}_{\mathbb{N}}(x) = x + 1$$

would remove both dependency pairs since there is a strict decrease for every cycle in the labeled dependency graph, which looks like



The problem in this example is that in the two dependency pairs the variable x does not correspond to the same term. For this example it is obvious that in any minimal non-terminating sequence, $\mathsf{s}(x)$ is substituted for the variable x in the second rule. Hence, one should not consider the original system but immediately change the variable x in the second rule on both sides to $\mathsf{s}(x)$. Then increasing interpretations are no longer successful. However such a transformation is not always possible. In the example above for every minimal non-terminating sequence there are no $\mathcal{R}$-steps and hence one can compute the substitution for x in the second rule by unification. Similar cases can be dealt with by instantiation and narrowing [3, 33].

To conclude, we state that increasing interpretations can be extended to allow an increase also in the variable part if the TRS under consideration satisfies two properties: (a) all dependency pairs are variable disjoint (this can always be achieved by renaming) and (b) for every minimal non-terminating sequence the $\mathcal{R}$-sequences are empty (and hence the values for variables can possibly be computed by unification). Note that one sufficient condition for (b) is that the set of usable rules is empty.

3.7 Summary

In this chapter a generalization of (polynomial) interpretations has been introduced. While traditional approaches must orient all rules at least weakly our approach allows an increase for some rules. This is possible due to a concurrent search for interpretations while considering the cycles in the dependency graph. Two DP processors have been proposed based on increasing interpretations. While the first one is a DP processor in the traditional sense, i.e., it allows to

remove dependency pairs (nodes from the dependency graph) the second one is able to eliminate single edges in the dependency graph. Two methods capable to implement increasing interpretations based on a transformation to arithmetic constraints have been proposed and evaluated on standard testbeds.

Chapter 4

Matrix Interpretations

Hofbauer and Waldmann proposed matrix interpretations in 2006 [41, 43] for SRSs. These contributions were (together with [10]) the first which considered SAT solvers to master the tremendously large search spaces in automated termination analysis of rewrite systems. In 2007 Endrullis *et al.* extended the method to TRSs [22]. Since then matrix interpretations became more and more popular due to their termination proving power. While the original setting just allows matrices with natural numbers as coefficients, arctic matrix interpretations also consider $-\infty$. They have been introduced in [51] based on the max-plus semiring and are especially powerful for SRSs. Furthermore a direct termination proof by arctic matrices (just possible for SRSs that may additionally contain constants) which orients all rules strictly implies linear derivational complexity. Other complexity results on (standard) matrices have been announced in [41] where it is shown that a TRS admitting a direct matrix proof may allow exponentially long derivations in the size of the starting term. Recently, [67] shows that triangular matrices are suitable to prove at most polynomially long derivations.

This chapter is organized as follows. In Section 4.1 matrix interpretations are extended to allow non-negative real coefficients. An empirical evaluation comparing matrix interpretations with coefficients from $\mathbb{N}$, $\mathbb{Q}$, and $\mathbb{R}$ is presented in Section 4.3. Section 4.4 discusses the benefits when allowing rational and real coefficients and compares our approach with related work.

4.1 Matrices over the Reals

In this section a DP processor based on matrix interpretations over the real numbers for TRSs is presented. In other words we combine matrix interpretations as presented in [22] with polynomials over rational and real coefficients [25, 62–64]. Especially the latter has so far only be used for polynomial interpretations. Although generalizing the theory from linear polynomial interpretations to matrices is straightforward, some tricks are needed to obtain an efficient implementation.

Formally, matrix interpretations are weakly monotone algebras $(\mathcal{M}, \geqslant, >)$ (see Definition 3.2 on page 39) where $\mathcal{M}$ is an $\mathcal{F}$-algebra over some carrier M^d for a fixed $d \in \mathbb{N}^{>0}$. In the sequel we consider $M = \mathbb{R}^{\geqslant 0}$. To fix the relations $\geqslant$ and $>$ that compare elements from M^d, i.e., vectors with non-negative real entries, we must fix how to compare elements from $\mathbb{R}^{\geqslant 0}$ first. The obvious candidate $>_{\mathbb{R}}$ is not suitable because it is not well-founded. But as

already suggested in earlier works for the case of polynomials [42, 62–64], one can approximate $>_{\mathbb{R}}$ by $>^{\delta}_{\mathbb{R}}$ defined as

$$x >^{\delta}_{\mathbb{R}} y \;:=\; x - y \geqslant_{\mathbb{R}} \delta$$

for $x, y \in \mathbb{R}$ and any $\delta \in \mathbb{R}^{>0}$. The next lemma shows that $>^{\delta}_{\mathbb{R}}$ has the desired property.

Lemma 4.1. *The order $>^{\delta}_{\mathbb{R}}$ is well-founded on $\mathbb{R}^{\geqslant 0}$ for any $\delta \in \mathbb{R}^{>0}$.*

Proof. Obvious. □

With the help of $>^{\delta}_{\mathbb{R}}$ it is now possible to define a well-founded order on vectors over M similar as in [22].

Definition 4.2. For vectors $\mathbf{u}$ and $\mathbf{v}$ from M^d we define

$$\begin{aligned} \mathbf{u} \geqslant \mathbf{v} \;&:=\; u_i \geqslant_{\mathbb{R}} v_i \text{ for } 1 \leqslant i \leqslant d, \text{ and} \\ \mathbf{u} >^{\delta} \mathbf{v} \;&:=\; u_1 >^{\delta}_{\mathbb{R}} v_1 \text{ and } u_i \geqslant_{\mathbb{R}} v_i \text{ for } 2 \leqslant i \leqslant d. \end{aligned} \tag{4.1}$$

Next the shape of the interpretations is fixed. For an n-ary function symbol $f \in \mathcal{F}^{\sharp}$ we define

$$f_{M^d}(\mathbf{x}_1, \ldots, \mathbf{x}_n) = F_1\mathbf{x}_1 + \cdots + F_n\mathbf{x}_n + \mathbf{f} \tag{4.2}$$

where $F_1, \ldots, F_n \in M^{d\times d}$ and $\mathbf{f} \in M^d$ if $f \in \mathcal{F}$ and $F_1, \ldots, F_n \in M^{1\times d}$ and $\mathbf{f} \in M$ if $f \in \mathcal{F}^{\sharp} \setminus \mathcal{F}$. As discussed in [22], using matrices of a different shape for dependency pair symbols reduces the search space while preserving the power of the method. Before addressing how to compare terms with respect to some interpretation we fix how to compare matrices. Let $m, n \in \mathbb{N}$. For $B, C \in M^{m\times n}$ we write

$$B \geqslant C \;:=\; B_{ij} \geqslant_{\mathbb{R}} C_{ij} \text{ for all } 1 \leqslant i \leqslant m,\ 1 \leqslant j \leqslant n.$$

Because of the linear shape of the interpretations, for a rule $l \to r$ with variables in $\{x_1, \ldots, x_k\}$ matrices $L_1, \ldots, L_k, R_1, \ldots, R_k$, and vectors $\mathbf{l}$ and $\mathbf{r}$ can be computed, such that

$$\begin{aligned} [\alpha]_{\mathcal{M}}(l) &= L_1\mathbf{x}_1 + \cdots + L_k\mathbf{x}_k + \mathbf{l} \text{ and} \\ [\alpha]_{\mathcal{M}}(r) &= R_1\mathbf{x}_1 + \cdots + R_k\mathbf{x}_k + \mathbf{r} \end{aligned}$$

where $\alpha(x) = \mathbf{x}$ for $x \in \mathcal{V}$. The next lemma states how to test $s >^{\delta}_{\mathcal{M}} t$ ($s \geqslant_{\mathcal{M}} t$), i.e., $[\alpha]_{\mathcal{M}}(s) >^{\delta} [\alpha]_{\mathcal{M}}(t)$ ($[\alpha]_{\mathcal{M}}(s) \geqslant [\alpha]_{\mathcal{M}}(t)$) for all assignments α effectively.

Lemma 4.3. *Let $l \to r$ be a rewrite rule and $[\alpha]_{\mathcal{M}}(l)$ and $[\alpha]_{\mathcal{M}}(r)$ be as above. Then for any $\delta \in \mathbb{R}^{>0}$*

- *$l \geqslant_{\mathcal{M}} r$ if and only if $L_i \geqslant R_i$ $(1 \leqslant i \leqslant k)$ and $\mathbf{l} \geqslant \mathbf{r}$,*
- *$l >^{\delta}_{\mathcal{M}} r$ if and only if $L_i \geqslant R_i$ $(1 \leqslant i \leqslant k)$ and $\mathbf{l} >^{\delta} \mathbf{r}$.*

Proof. Immediate from the proof of Lemma 4 in [22]. □

Next we show that $(\mathcal{M}, \geqslant, >^\delta)$ may be used for termination proofs within the dependency pair setting.

Theorem 4.4. *Let $\mathcal{F}$ be signature, $M = \mathbb{R}^{\geqslant 0}$, and $\mathcal{M}$ be an $\mathcal{F}$-algebra over the carrier M^d for some $d \in \mathbb{N}^{>0}$ with f_{M^d} of the shape as in (4.2) for all $f \in \mathcal{F}$. Then for any $\delta \in \mathbb{R}^{>0}$ the algebra $(\mathcal{M}, \geqslant, >^\delta)$ is weakly monotone.*

Proof. According to Definition 3.2 we have to show that every interpretation function is monotone with respect to $\geqslant$ which is obvious. Next we show that the relation $>^\delta$ is well-founded. From (4.1) it is obvious that $>^\delta$ is well-founded (on the carrier M^d) since $>^\delta_{\mathbb{R}}$ is well-founded on $\mathbb{R}^{\geqslant 0}$ for any $\delta \in \mathbb{R}^{>0}$. The latter holds by Lemma 4.1. The last condition for a weakly monotone algebra is compatibility, i.e., $>^\delta \cdot \geqslant \subseteq >^\delta$ which trivially holds. □

Due to Theorem 3.3 we get the following corollary.

Corollary 4.5. *If $(\mathcal{M}, \geqslant, >^\delta)$ is a weakly monotone algebra then $(\geqslant_{\mathcal{M}}, >^\delta_{\mathcal{M}})$ is a reduction pair.* □

The concepts introduced so far are demonstrated by the following example.

Example 4.6. Consider the TRS from Example 2.27 on page 24 again. One proof obligation is the DP problem $(\mathcal{P}, \mathcal{R})$ where $\mathcal{P}$ consists of the rule

$$\mathsf{MINUS}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{MINUS}(x, y)$$

and $\mathcal{R}$ consists of the rules

$$\begin{array}{ll} \mathsf{minus}(x, \mathsf{0}) \to x & \mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) \to \mathsf{0} \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{minus}(x, y) & \mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))). \end{array}$$

Using dimension 2 and $\delta = 1$ the interpretation

$$\begin{array}{ll} \mathsf{MINUS}_{M^2}(\mathbf{x}, \mathbf{y}) = \begin{pmatrix} 1 & 0 \end{pmatrix} \mathbf{x} & \mathsf{minus}_{M^2}(\mathbf{x}, \mathbf{y}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \mathbf{x} \\ \mathsf{s}_{M^2}(\mathbf{x}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \mathbf{x} + \begin{pmatrix} \sqrt{2} \\ 0 \end{pmatrix} & \mathsf{quot}_{M^2}(\mathbf{x}, \mathbf{y}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \mathbf{x} \\ \mathsf{0}_{M^2} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \end{array}$$

orients the dependency pair strictly and the other rules weakly as can be seen below (where t abbreviates $\mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y)))$):

$$[\mathsf{MINUS}(\mathsf{s}(x), \mathsf{s}(y))]_{\mathcal{M}} = \begin{pmatrix} 1 & 0 \end{pmatrix} \mathbf{x} + \sqrt{2} >^1_{\mathcal{M}} \begin{pmatrix} 1 & 0 \end{pmatrix} \mathbf{x} = [\mathsf{MINUS}(x, y)]_{\mathcal{M}}$$

$$[\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y))]_{\mathcal{M}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \mathbf{x} + \begin{pmatrix} \sqrt{2} \\ 0 \end{pmatrix} \geqslant_{\mathcal{M}} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \mathbf{x} = [\mathsf{minus}(x, y)]_{\mathcal{M}}$$

$$[\mathsf{minus}(\mathsf{0}, y)]_{\mathcal{M}} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \geqslant_{\mathcal{M}} \begin{pmatrix} 0 \\ 0 \end{pmatrix} = [\mathsf{0}]_{\mathcal{M}}$$

$$[\mathsf{quot}(\mathsf{0}, \mathsf{s}(y))]_{\mathcal{M}} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \geqslant_{\mathcal{M}} \begin{pmatrix} 0 \\ 0 \end{pmatrix} = [\mathsf{0}]_{\mathcal{M}}$$

$$[\mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y))]_{\mathcal{M}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \mathbf{x} + \begin{pmatrix} \sqrt{2} \\ 0 \end{pmatrix} \geqslant_{\mathcal{M}} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \mathbf{x} + \begin{pmatrix} \sqrt{2} \\ 0 \end{pmatrix} = [t]_{\mathcal{M}}$$

To conclude the example we show that above interpretation strictly orients the dependency pair $\mathsf{MINUS}(\mathsf{s}(x),\mathsf{s}(y)) \to \mathsf{MINUS}(x,y)$. Due to Lemma 4.3 $\begin{pmatrix}1 & 0\end{pmatrix}\mathbf{x} + \sqrt{2} >^1_{\mathcal{M}} \begin{pmatrix}1 & 0\end{pmatrix}\mathbf{x}$ if and only if $\begin{pmatrix}1 & 0\end{pmatrix} \geqslant \begin{pmatrix}1 & 0\end{pmatrix}$ and $\sqrt{2} >^1_{\mathbb{R}} 0$. The former is obviously satisfied and the latter holds since $\sqrt{2} - 0 \geqslant_{\mathbb{R}} 1$.

Next we address the problem to find a suitable δ automatically. Since δ influences if a rule can be oriented strictly or not it cannot be chosen arbitrarily. E.g., the interpretation from Example 4.6 with $\delta = 2$ can no longer orient the rule $\mathsf{MINUS}(\mathsf{s}(x),\mathsf{s}(y)) \to \mathsf{MINUS}(x,y)$ strictly since $\sqrt{2} \not>^2_{\mathbb{R}} 0$. For DP problems containing only finitely many rules (which corresponds to the usual setting) a suitable δ can easily be computed. The reason is that for such DP problems only finitely many rules are involved in the strict comparison, i.e., to test for a rule $s \to t$ if $s >^\delta_{\mathcal{M}} t$ the comparison $\mathbf{s} >^\delta \mathbf{t}$ is needed (cf. Lemma 4.3) which boils down to $s_1 >^\delta_{\mathbb{R}} t_1$ (cf. (4.1)). Since $s_1 - t_1 \geqslant_{\mathbb{R}} \delta$ is tested for only finitely many rules $s \to t$, the minimum of all $s_1 - t_1$ is well-defined and provides a suitable δ. But the next lemma (which generalizes the idea from [62] to matrices) states that actually there is no need to compute δ explicitly.

Lemma 4.7. *Let $(\mathcal{P},\mathcal{R})$ be a DP problem. If $\mathcal{P}$ contains finitely many rules then δ need not be computed explicitly.*

Proof. Due to the discussion above one can obtain a $\delta \in \mathbb{R}^{>0}$ such that for every $s \to t \in \mathcal{P}$ we have $s_1 >^\delta_{\mathbb{R}} t_1$ if and only if $s_1 >_{\mathbb{R}} t_1$. Hence for all strict comparisons that occur the relations $>^\delta_{\mathbb{R}}$ and $>_{\mathbb{R}}$ coincide. Consequently it is safe if an implementation uses $>_{\mathbb{R}}$ instead of $>^\delta_{\mathbb{R}}$ and ignores the exact δ. □

4.2 Implementation

This section first sketches how to encode matrix interpretations as arithmetic constraints. Then it addresses optimizations that are essential when matrices of larger dimensions are considered for non-natural matrix coefficients.

Implementing a DP processor based on matrix interpretations is a search problem. After fixing the dimension d for every n-ary function symbol $f \in F^\sharp$ matrices $F_1,\dots,F_n$ and a vector $\mathbf{f}$ must be computed such that some constraints (see below) are satisfied. The coefficients of these matrices are existentially quantified arithmetic variables. Lifting addition and multiplication from coefficients to matrices as usual allows to interpret terms. The term interpretations can then be compared using Lemma 4.3, yielding encodings for $\geqslant_{\mathsf{mat}}$ and $>_{\mathsf{mat}}$. These encodings are parametrized by the dimension d.

Since matrix interpretations have argument filterings automatically built-in (a zero matrix as coefficient corresponds to a deleting position in the argument filtering) we sketch an encoding that implements Theorem 1.10 (without an explicit argument filtering) based on matrix interpretations. The aim is to define a formula

$$\mathsf{U}(\mathcal{P},\mathcal{R},\geqslant_{\mathsf{mat}}) \;\wedge \bigvee_{s\to t\in\mathcal{P}} s >_{\mathsf{mat}} t$$

the satisfiability of which ensures $\mathcal{U}_{\mathcal{M}}(\mathcal{P},\mathcal{R}) \cup \mathcal{P} \subseteq \geqslant_{\mathcal{M}}$ and $\mathcal{P} \cap >_{\mathcal{M}} \neq \varnothing$ for some weakly monotone algebra $(\mathcal{M},\geqslant,>)$ based on matrix interpretations.

Here $\mathcal{U}_{\mathcal{M}}(\mathcal{P},\mathcal{R})$ are the usable rules with respect to the interpretation. The only difference to $\mathcal{U}_{\pi}(\mathcal{P},\mathcal{R})$ is that here it is demanded that the interpretation depends on the argument (coefficient is non-zero) instead of demanding that the argument filtering π keeps this argument. The formula $\mathsf{U}(\mathcal{P},\mathcal{R},\succcurlyeq_{\mathsf{mat}})$ is defined similarly to the one for KBO on page 25 as follows:

$$\bigwedge_{l\to r\in\mathcal{P}} \big(U_{\mathrm{root}(l)} \wedge l \succcurlyeq_{\mathsf{mat}} r\big) \wedge \bigwedge_{l\to r\in\mathcal{R}} \big(U_{\mathrm{root}(l)} \to l \succcurlyeq_{\mathsf{mat}} r\big)$$

and

$$\bigwedge_{l\to r\,\in\,\mathcal{R}\cup\mathcal{P}} \Bigg(U_{\mathrm{root}(l)} \to \bigwedge_{\substack{p\,\in\,\mathcal{P}\mathrm{os}_{\mathcal{F}}(r)\\ \mathrm{root}(r|_p)\text{ is defined}}} \Bigg(\bigwedge_{q,\,i\colon\, iq\leqslant p} \neg\,\mathsf{zero}(\mathrm{root}(r|_q),i) \to U_{\mathrm{root}(r|_p)}\Bigg)\Bigg)$$

where

$$\mathsf{zero}(f,i) \;=\; \bigwedge_{\substack{1\leqslant j\leqslant m\\ 1\leqslant k\leqslant n}} \big((F_i)_{jk}=0\big).$$

Here U_f is a propositional variable indicating if rules with root f must be weakly oriented (second conjunct). The encoding demands that all rules from $\mathcal{P}$ must be oriented (first conjunct) and a rule in $\mathcal{R}$ must be considered if its root may affect the interpretation of some right-hand side of a rule that must be oriented (third conjunct).

Next we address how to represent the matrix coefficients. Natural numbers are considered in binary notation and expressed as bit-vectors. Rational coefficients are implemented as described in Section 6.1.3, i.e., the numerator is some bit-vector representing a number and the denominator is a fixed positive integer. Real numbers are encoded as pairs consisting of a non-real and a real part. For details see Section 6.1.4.

Some relevant issues concerning rational numbers are discussed next. Since every coefficient has the same denominator no heuristics must be applied to decide which coefficient might take non-integral values. On the other hand consecutive multiplications of such fractions give rise to an exponential explosion of the denominator. Due to the large denominators many bits are needed to represent the numerator in binary which turned out to be the main bottleneck. Next we demonstrate this problem by means of a (strongly simplified) example and propose an elegant escape immediately afterwards.

To ease readability—in contrast to the actual implementation—the examples within this section have concrete numbers as numerators. This does not affect the problems that arise. When computing the interpretation of a term various additions and multiplications are needed. Hence expressions like in the following example must be computed.

Example 4.8. Without canceling (intermediate) results we get

$$\left(\frac{1}{2}\times\frac{4}{2}\right)\times\frac{3}{2}+\frac{1}{2} = \frac{4}{4}\times\frac{3}{2}+\frac{1}{2} = \frac{12}{8}+\frac{1}{2} = \frac{16}{8}.$$

The main reason why we did not cancel the fractions in the example above is that the same happens in the implementation. Due to the fact that the numerator is some bit-vector consisting of propositional formulas its concrete value is unknown and hence no cancellation is possible. Since the encodings of matrices produce much larger constraints than the one above the numerators grow rapidly. We propose the following elegant escape which is very easy to implement and has positive effects on the run-time (cf. Section 4.3). We force that a fraction is canceled if the denominator exceeds some given value. The next example shows the positive and negative aspects of this heuristic.

Example 4.9. First we demonstrate the positive aspects. To this end we allow a maximal denominator of at most 2. Thus after every addition or multiplication the fraction is canceled whenever the denominator exceeds 2. The single steps of the computation involving this heuristic are given below:

$$\left(\frac{1}{2} \times \frac{4}{2}\right) \times \frac{3}{2} + \frac{1}{2} = \frac{2}{2} \times \frac{3}{2} + \frac{1}{2} = \frac{3}{2} + \frac{1}{2} = \frac{4}{2}$$

The negative aspects become apparent if the denominator is chosen too small. Then some computations can no longer be performed, e.g., when allowing a denominator of 1 the computation gets stuck in the second step since $\frac{3}{2}$ cannot be canceled, as can be seen in the sequence below:

$$\left(\frac{1}{2} \times \frac{4}{2}\right) \times \frac{3}{2} + \frac{1}{2} = \frac{1}{1} \times \frac{3}{2} + \frac{1}{2} = \frac{?}{1} + \frac{1}{2}$$

In the implementation, canceling by two is achieved by dividing the denominator by two and dropping the least significant bit of the numerator while demanding that this bit evaluates to false. Hence in contrast to the example above computations do not get stuck, but may produce unsatisfiable formulas. The next section shows that this does not happen very frequently. Furthermore, there also the effectiveness of this very simple but efficient heuristic is demonstrated.

4.3 Experiments

All tables within this chapter have been produced with TTT2 using the SAT back-end for solving arithmetic constraints (cf. Section 6.1). Due to the fact that matrix interpretations need non-linear arithmetic, this is the only setting supported. The exact strategies to call TTT2 are listed in Section A.4. Next we compare the performance of matrix interpretations where matrices have different coefficients, ranging from $\mathbb{N}$ via $\mathbb{Q}$ to $\mathbb{R}$. For TRSs, the coefficients of a matrix over dimension d are represented in $\max\{2, 5-d\}$ bits (for reals we allow $\max\{1, 3-d\}$ bits due to the more expensive pair representation, cf. Section 6.1.4). Every rational coefficient is represented as a fraction with fixed denominator 2. Hence a matrix of dimension two admits natural coefficients $\{0, 1, \ldots, 7\}$, rational coefficients $\{0, \frac{1}{2}, 1, 1\frac{1}{2}, 2, 2\frac{1}{2}, 3, 3\frac{1}{2}\}$, and real coefficients $\{0, 1, \sqrt{2}, 1+\sqrt{2}\}$. The number of bits for representing intermediate computations was chosen to be one more than the number of bits allowed for the

Table 4.1: Matrices with dependency pairs for 1391 TRSs

	1×1			2×2			3×3		
	yes	time	t/o	yes	time	t/o	yes	time	t/o
$\mathbb{N}$	545	8885	83	618	23820	326	627	25055	349
$\mathbb{Q}$	599	8574	67	597	20238	261	496	19490	252
$\mathbb{Q}_1$	606	5906	46	655	15279	173	643	14062	164
$\mathbb{Q}_2$	627	10109	93	651	23102	308	619	23806	330
$\mathbb{R}$	535	17029	198	630	16517	200	599	29346	415

Table 4.2: Matrices with dependency pairs for 732 SRSs

	1×1			2×2			3×3		
	yes	time	t/o	yes	time	t/o	yes	time	t/o
$\mathbb{N}$	69	1795	8	103	20869	225	110	37458	576
$\mathbb{Q}$	62	1089	3	78	10533	81	52	19066	257
$\mathbb{Q}_1$	84	1136	3	121	13603	125	117	31040	424
$\mathbb{Q}_2$	92	1717	6	127	18241	193	113	34816	518
$\mathbb{R}$	58	11256	116	77	31195	382	54	41589	676

coefficients. Restricting the bit-width of computations is essential for performance, especially for matrices of larger dimensions. For SRSs we allow one additional bit for both matrix coefficients and intermediate results.

In Tables 4.1 and 4.2 matrices from dimensions one to three are considered. The rows labeled $\mathbb{N}$ indicate that only natural numbers are allowed as coefficients whereas $\mathbb{Q}$ refers to the naive representation of rationals without canceling the fractions (for details cf. Section 6.1.3) and $\mathbb{R}$ to algebraic coefficients (cf. Section 6.1.4). Performance of $\mathbb{Q}$ is satisfactory for matrices with dimension one (which correspond to linear polynomial interpretations and confirms the results in [25]) but poor for larger dimensions due to the reasons discussed in the previous section. The rows $\mathbb{Q}_n$ correspond to the setting where a fraction is canceled if its denominator exceeds n. The column labeled yes shows the number of successful termination proofs, while time indicates the total time needed by the tool in seconds. If no answer was produced within 60 seconds the execution is killed (column t/o). From these numbers it becomes apparent that rational coefficients indeed increase termination proving power of matrix interpretations in practice. Interestingly the overall performance of $\mathbb{Q}_1$ is surprisingly good where $\mathbb{Q}_1$ just allows rational values for the matrix coefficients but not for any intermediate results. The (optimized) approach allowing rationals does not only increase the total number of systems proved terminating but additionally reports these results faster than the comparable approach based on natural numbers. Hence only the benefits of allowing rational numbers remain as can be witnessed in Tables 4.1 and 4.2.

4.4 Assessment

The experimental results from the previous section show that a naive implementation of rational numbers does not pay off for matrices of larger dimensions. Hence we developed a heuristic which is very easy to implement while providing substantial gains in efficiency and termination proving power. Furthermore due to the design of the constraint solving module (cf. Chapter 6) no changes have been necessary within the encoding of the matrix method when going from natural to rational and real coefficients. And if one looks beyond TPDB, then our implementation can also show its strength for real coefficients. It masters the TRS $\mathcal{R}_{\mathbb{R}}$ from Example 4.10 below. This system stems from [63] where it was proved that no direct termination proof based on polynomial interpretations over the natural or rational numbers can exist which orients all rules strictly. However a proof over the reals is possible and our implementation finds such a proof fully automatically. We must confess that the interpretation below is not the first one given by an automatic termination analyzer since Mu-term [61] is also capable of reasoning about the real numbers. However, Mu-term's constraint solving mechanism is not based on SAT which makes our contribution unique.

Example 4.10. For the TRS $\mathcal{R}_{\mathbb{R}}$ consisting of the seven rules

$$\begin{aligned} \mathsf{k}(x,x,\mathsf{b}_1) &\to \mathsf{k}(\mathsf{g}(x),\mathsf{b}_2,\mathsf{b}_2) & \mathsf{g}(\mathsf{c}(x)) &\to \mathsf{f}(\mathsf{c}(\mathsf{f}(x))) \\ \mathsf{k}(x,\mathsf{a}_2,\mathsf{b}_1) &\to \mathsf{k}(\mathsf{a}_1,x,\mathsf{b}_1) & \mathsf{f}(\mathsf{f}(x)) &\to \mathsf{g}(x) \\ \mathsf{k}(\mathsf{a}_4,x,\mathsf{b}_1) &\to \mathsf{k}(x,\mathsf{a}_3,\mathsf{b}_1) & \mathsf{f}(\mathsf{f}(\mathsf{f}(\mathsf{f}(x)))) &\to \mathsf{k}(x,x,x) \\ \mathsf{k}(\mathsf{g}(x),\mathsf{b}_3,\mathsf{b}_3) &\to \mathsf{k}(x,x,\mathsf{b}_4) \end{aligned}$$

TTT2 finds the interpretation that orients all rules strictly

$$\begin{aligned} \mathsf{a}_{1\mathbb{R}} &= 0 & \mathsf{b}_{1\mathbb{R}} &= 2+\sqrt{2} & \mathsf{f}_{\mathbb{R}}(x) &= \sqrt{2}x+\sqrt{2} \\ \mathsf{a}_{2\mathbb{R}} &= 1+2\sqrt{2} & \mathsf{b}_{2\mathbb{R}} &= 0 & \mathsf{g}_{\mathbb{R}}(x) &= 2x+1+\sqrt{2} \\ \mathsf{a}_{3\mathbb{R}} &= 0 & \mathsf{b}_{3\mathbb{R}} &= 1+\sqrt{2} & \mathsf{c}_{\mathbb{R}}(x) &= x+1+2\sqrt{2} \\ \mathsf{a}_{4\mathbb{R}} &= 1+\sqrt{2} & \mathsf{b}_{4\mathbb{R}} &= \sqrt{2} & \mathsf{k}_{\mathbb{R}}(x,y,z) &= x+y+\sqrt{2}z+3\sqrt{2} \end{aligned}$$

within a fraction of a second. However while a direct proof with polynomials over $\mathbb{N}$ is not possible, natural coefficients suffice in the dependency pair setting (after computing the SCCs of the dependency graph). Consequently all contemporary termination tools can prove this system terminating.

We conclude this section with a comparison to related work. The idea of allowing rational numbers for matrix interpretations is not new. Already in 2007 a first proposal to consider matrices (for termination proofs of SRSs) with rational coefficients emerged [27]. In this note evolutionary algorithms [85] are suggested to tackle the problem how to find suitable rational coefficients.

In [25] polynomial interpretations are extended to rational coefficients. This work is related since linear polynomial interpretations coincide with matrix interpretations of dimension one. Our experiments confirm the gains in power

when using matrices of dimension one. The implementation suggested in [25] naturally extends to matrices of larger dimensions but the discussion from the previous section shows that this results in a poor performance without further ado.

Independently to our research the authors in [1] recently extended the theory of matrix interpretations to coefficients over the reals. However, in contrast to TTT2 their (preliminary) implementation can only deal with rationals. Furthermore no benchmarks are given in [27, 1] showing any gains in power by allowing rationals. Hence the results from this chapter for the first time give evidence that matrix interpretations over the non-negative reals do really allow to extend the power of termination provers in practice.

4.5 Summary

This chapter dealt with extending matrix interpretations over the natural numbers to non-negative reals. We proved that matrices over the reals yield reduction pairs which makes them applicable for termination proofs within the dependency pair setting. While for small dimensions rational coefficients substantially increase the yes-score, for higher dimensions the approach does not pay off in the naive setting. There the key issue is to keep the values of numerators moderate by canceling the fraction if the denominator exceeds some given limit. Last but not least, unlike many other implementations, we showed that our tool is capable of finding matrix interpretations over the reals (not only rationals). To the author's knowledge, TTT2 is the first tool supporting reasoning about algebraic numbers with the help of SAT solvers.

Chapter 5

Loops

Surprisingly—compared to the vast amount of methods devoted to proving termination—only few techniques concerning non-termination are known and implemented. Nevertheless checking for non-termination is extremely useful, for instance when debugging programs since methods for non-termination provide evidence (e.g., a loop) why a system is not terminating, and a concrete counterexample is helpful to track down a bug. Most non-trivial approaches in that direction aim to find *looping* reductions and comprise ancestor graphs [98], narrowing [32], match-bounds [83], unfoldings [71], and transport systems [84]. The first automated approach [70] dealing with non-looping non-terminating systems was presented during the 2008 edition of the termination competition.

The increasing complexity of proofs generated by termination tools makes automated certification of their output more and more important. Since 2007 a certified category is part of the termination competition. Here, the participating tools are required to generate proofs that can automatically be certified. Current approaches for automatic certification of termination proofs are Coccinelle/C*i*ME [12, 14], CoLoR/Rainbow [6], and IsaFoR/CeTA [77, 80]. The first two projects use Coq [5] as theorem prover. Here, Coccinelle and CoLoR are Coq-libraries on rewriting whereas C*i*ME and Rainbow transform proof output of some termination tool into Coq-script using the respective library. Afterwards, Coq is used to certify the result. IsaFoR/CeTA uses Isabelle/HOL[1] as theorem prover. IsaFoR (*Isa*belle *F*ormalization *o*f *R*ewriting) is an Isabelle-library on rewriting and CeTA (*Ce*rtified *T*ermination *A*nalysis) is a Haskell program that certifies proof output of some termination tool directly (without calling a theorem prover). It is generated from IsaFoR using Isabelle's code-generation facilities [36]. We extend IsaFoR/CeTA by a loop-checker capable to certify loops which makes it the first certifier supporting non-termination techniques.

The remainder of this chapter is organized as follows: In Section 5.1 we present a novel method to find loops for the special case of string rewriting. Afterwards, Sections 5.2 and 5.3 discuss our Isabelle formalization for IsaFoR and our check-functions for CeTA that are used to certify looping non-termination. Experimental results together with an assessment of our contributions can be found in Section 5.4 before future work is addressed in Section 5.5.

This chapter heavily contributes to [93].

[1] In the remainder we use Isabelle instead of Isabelle/HOL.

5.1 Finding Loops for String Rewrite Systems

In this section we consider SRSs only. Instead of $\mathsf{a}(\mathsf{b}(\mathsf{c}(x)))$ we write abc (i.e., the variable is implicit). For a string s we denote the i-th symbol ($1 \leqslant i \leqslant \|s\|$) in s by s_i, e.g., $\mathsf{abc}_2 = \mathsf{b}$. An SRS $\mathcal{S}$ is *length-preserving* if for every rule $l \to r \in \mathcal{S}$ the condition $|l| = |r|$ holds.

Example 5.1. Consider the SRS $\mathcal{S} = \{\mathsf{ab} \to \mathsf{bbaa}\}$ which admits the looping reduction

$$\mathsf{abb} \to \mathsf{bbaab} \to \mathsf{bbabbaa}$$

where the initial string abb is reached again after two rewrite steps wrapped in the context $C = \mathsf{bb}\Box$ and instantiated by the substitution $\sigma = \{x/\mathsf{aa}\}$. Hence $\mathcal{S}$ admits the loop $([\mathsf{abb}, \mathsf{bbaab}], C, \sigma)$ of length 2.

The main benefit of the dependency pair approach (for finding loops) is that leading contexts as in Example 5.1 are automatically removed by construction of the dependency pairs [32], and as a result a looping reduction in a DP problem $(\mathcal{P}, \mathcal{R})$ takes the form $t \to^+_{\mathcal{P}\cup\mathcal{R}} t\sigma$. Our idea is to encode a looping rewrite sequence within the DP framework using a matrix of dimension $m \times n$ where $(0, 0)$ denotes the top left entry and $(m-1, n-1)$ the bottom right one. Every row in the matrix corresponds to a string and the intended meaning is that there is a rewrite step from row i to row $i + 1$.

Example 5.2. Consider again the SRS from Example 5.1 which gives rise to the dependency pairs $\mathsf{Ab} \to \mathsf{Aa}$ and $\mathsf{Ab} \to \mathsf{A}$. Using a 3×5 matrix a looping reduction is possible. The entries marked with $\cdot$ indicate that any symbol might appear at these positions.

$$\begin{array}{ccccc} \mathsf{A} & \mathsf{b} & \mathsf{b} & \cdot & \cdot \\ \mathsf{A} & \mathsf{a} & \mathsf{b} & \cdot & \cdot \\ \mathsf{A} & \mathsf{b} & \mathsf{b} & \mathsf{a} & \mathsf{a} \end{array}$$

In the sequel we describe how to represent such matrices for a DP problem $(\mathcal{P}, \mathcal{R})$ in propositional logic where the following variables are used:[2]

M^a_{ij}	symbol a occurs at position (i, j) of the matrix
$R^{l\to r}_i$	in row i of the matrix rule $l \to r$ is applied
p_i	the position (= column) in row i where the rule is applied (in Example 5.2 we have $\mathrm{p}_0 = 0$ and $\mathrm{p}_1 = 1$)
e_i	pointing to the last symbol of the i-th string (in the example $\mathrm{e}_0 = 2$, $\mathrm{e}_1 = 2$, and $\mathrm{e}_2 = 4$)

The variables p_i and e_i are not Boolean but represent natural numbers. To distinguish them from proper propositional variables they are denoted in lower case.

[2] The idea of encoding computation as propositional satisfiability goes back to [13].

Exactly one function symbol: To get exactly one function symbol at each matrix position, we ensure at least one symbol per entry and additionally ban multiple symbols at the same entry. Note that dependency pair symbols (those in $\mathcal{F}^\sharp \setminus \mathcal{F}$) can only appear at column 0 of each row. This is encoded as follows (where $X = \mathcal{F}$ if $j > 0$ and $X = \mathcal{F}^\sharp \setminus \mathcal{F}$ otherwise):

$$\alpha_{ij} \;=\; \bigvee_{a \in X} M^a_{ij} \;\wedge\; \bigwedge_{a \in X} \big(M^a_{ij} \to \bigwedge_{b \in X \setminus \{a\}} \neg M^b_{ij}\big)$$

Rule application: If a rule $l \to r$ applies in row i ($R_i^{l \to r}$) the rule must be applied correctly ($\mathsf{app}_i^{l \to r}$) and entries unaffected by the rule application must be copied from row i to row $i+1$ ($\mathsf{cp}_{ij}^{l \to r}$). The position of the rule application is fixed by p_i and satisfying $\mathsf{cp}_{ij}^{l \to r}$ has the side effect that only one rule is applied. Hence

$$\beta_i^{l \to r} \;=\; R_i^{l \to r} \;\to\; \mathsf{app}_i^{l \to r} \wedge \bigwedge_{0 \leqslant j < n} \mathsf{cp}_{ij}^{l \to r}$$

where in case of $l \to r \in \mathcal{R}$ we have

$$\begin{aligned} \mathsf{app}_i^{l \to r} \;=\; & \bigwedge_{0 \leqslant j < \|l\|} M^{l_{j+1}}_{i(\mathrm{p}_i + j)} \wedge \bigwedge_{0 \leqslant j < \|r\|} M^{r_{j+1}}_{(i+1)(\mathrm{p}_i + j)} \\ & \wedge\; \mathrm{e}_{i+1} + \|l\| = \mathrm{e}_i + \|r\| \;\wedge\; \mathrm{e}_i \geqslant \mathrm{p}_i + \|l\| \end{aligned}$$

and if $l \to r \in \mathcal{P}$ then p_i specializes to 0. The first (second) conjunct of $\mathsf{app}_i^{l \to r}$ applies the left-hand side in row i (right-hand side in row $i+1$) at the abstract position p_i. The last but one conjunct demands that the end pointer in line $i+1$ takes the value of $\mathrm{e}_i - \|l\| + \|r\|$. To ensure that the contracted redex fits the string in line i the last conjunct must be satisfied.

The formula for $\mathsf{cp}_{ij}^{l \to r}$ is defined as $\top$ if $j + \max\{\|l\|, \|r\|\} \geqslant n$ (these entries would be outside of the matrix), as

$$\big(j < \mathrm{p}_i \wedge \bigwedge_{a \in X} (M^a_{ij} \leftrightarrow M^a_{(i+1)j})\big) \;\vee\; \big(j \geqslant \mathrm{p}_i \wedge \bigwedge_{a \in \mathcal{F}} (M^a_{i(j+\|l\|)} \leftrightarrow M^a_{(i+1)(j+\|r\|)})\big)$$

(where $X = \mathcal{F}^\sharp \setminus \mathcal{F}$ if $j = 0$ and $X = \mathcal{F}$ otherwise) if $l \to r \in \mathcal{R}$, and as

$$\bigwedge_{a \in \mathcal{F}} \big(M^a_{i(j+\|l\|)} \leftrightarrow M^a_{(i+1)(j+\|r\|)}\big)$$

if $l \to r \in \mathcal{P}$.

All entries in the matrix before the position where the rule is applied are copied from row i to $i+1$. The second disjunct copies the entries after p_i which are unaffected when applying the rule. The positions of these entries change if the applied rule is not length-preserving. For rules $l \to r \in \mathcal{P}$ we know that $\mathrm{p}_i = 0$ and hence the formula simplifies to the one shown above.

Initial string is reached again: To recognize a loop, the string in some row $i > 0$ has to match the one in row zero. Furthermore the end pointer for this row is not allowed to be smaller than the one of row zero.

$$\gamma \;=\; \bigvee_{0<i<m} \Bigg(\bigwedge_{a \in \mathcal{F}^\sharp \setminus \mathcal{F}} \big(M^a_{00} \leftrightarrow M^a_{i0}\big) \;\wedge \bigwedge_{\substack{0<j<n \\ a \in \mathcal{F}}} \big(M^a_{0j} \leftrightarrow M^a_{ij}\big) \;\wedge\; \mathrm{e}_i \geqslant \mathrm{e}_0 \Bigg)$$

Putting everything together: For a DP problem $(\mathcal{P}, \mathcal{R})$ the formula $\mathsf{loop}^{m,n}_{\mathcal{P},\mathcal{R}}$ is defined as

$$\bigwedge_{0 \leqslant i < m} \Bigg(\bigwedge_{0 \leqslant j < n} \alpha_{ij} \;\wedge\; \mathrm{e}_i < n \;\wedge\; \beta_i \Bigg) \;\wedge\; \gamma$$

with

$$\beta_i \;=\; \bigvee_{l \to r \in \mathcal{P} \cup \mathcal{R}} R_i^{l \to r} \;\wedge \bigwedge_{l \to r \in \mathcal{P} \cup \mathcal{R}} \beta_i^{l \to r}$$

expressing that one rule has to apply in row i (and that it is applied properly). The condition $\mathrm{e}_i < n$ ensures that all strings in the loop stay within the allowed matrix dimensions.

There are two types of variables keeping track of function symbols—concrete (M^a_{ij}) and abstract ones (M^a_{ix}) where x is an arithmetical expression representing a natural number. The latter are needed when a rule is applied at the abstract position p_i. In the current encoding, abstract variables M^{a}_{3x} and M^{a}_{3y} denote two different expressions (since the expressions x and y differ) and hence may take different values. If the expressions x and y evaluate to the same number, we want to enforce that the variables take identical values. In the implementation we test for every such abstract variable M^a_{ix} whether it matches a concrete one M^a_{ij} and we identify them if that is the case in order to obtain consistent results:

$$\varphi_{\mathsf{cons}} = \bigwedge_{M^a_{ix}} \bigwedge_{0 \leqslant j < n} \big(x = j \to (M^a_{ij} \leftrightarrow M^a_{ix})\big)$$

Next the main theorem for finding loops is formulated.

Theorem 5.3. *A DP problem $(\mathcal{P}, \mathcal{R})$ admits a loop of length at most m involving strings of size at most $n+1$ if the formula $\mathsf{loop}^{m+1,n}_{\mathcal{P},\mathcal{R}} \wedge \varphi_{\mathsf{cons}}$ is satisfiable.* □

The above theorem allows to implement a DP processor for non-termination similar to the one in [32, Theorem 26] of the following shape.

Theorem 5.4. *The DP processor that returns for a DP problem $(\mathcal{P}, \mathcal{R})$ "no" if $\mathcal{P} \cup \mathcal{R}$ admits a loop and $\{(\mathcal{P}, \mathcal{R})\}$ otherwise is sound and complete.* □

Although the above DP processor is complete, the approach is incomplete in a different sense. In [32] it is shown that a TRSs $\mathcal{R}$ is looping if and only if the DP problem $(\mathsf{DP}(\mathcal{R}),\mathcal{R})$ admits a loop with empty context. But this loop need not be *minimal*. The following example shows that there exist TRSs that are looping but only admit non-looping minimal sequences.

Example 5.5. Consider the SRS (from [16]) consisting of the following three rules

$$\mathsf{bc} \to \mathsf{dc} \qquad \mathsf{bd} \to \mathsf{db} \qquad \mathsf{ad} \to \mathsf{abb}$$

which was proved non-looping in [29]. Nevertheless it admits a non-terminating sequence since for every $i > 0$ we have $\mathsf{ab}^i\mathsf{c} \to^+ \mathsf{ab}^{i+1}\mathsf{c}$. Now consider the single rule $\mathsf{eabc} \to \mathsf{eabc}$ which obviously allows a looping reduction. Let $\mathcal{S}$ denote the union of all four rules. Then using the DP problem $(\mathsf{DP}(\mathcal{S}), \mathcal{S})$, the term Eabc does not start a looping *minimal* sequence since the subterm abc is not terminating with respect to $\mathcal{S}$. Furthermore one can easily verify that actually all minimal infinite sequences are non-looping. However, our encoding from Theorem 5.3 is "complete" in the sense that it easily finds the looping reduction $\mathsf{Eabc} \to_{\mathsf{DP}(\mathcal{S})\cup\mathcal{S}} \mathsf{Eabc}$.

Next we show that any loop found within the DP framework can be transformed into a loop in the original rewrite system. This is of interest since the formalization in the next section does only support such loops. We do not expect major difficulties for lifting the formalization of loops into the DP setting but due to Lemma 5.7 below we see no need in doing so. Besides, for the purpose of debugging, it is more useful to get a counterexample in the original system than in some transformed system (where the connection between the counterexample and the corresponding bug may not be obvious). Furthermore, to certify loops within the DP setting, all DP processors employed before the loop processor must be formalized and proved complete (whereas for termination proofs soundness suffices).

The next example gives the intuition how loops are transformed.

Example 5.6. Consider the loop

$$\mathsf{Abb} \to_{\mathsf{DP}(\mathcal{S})} \mathsf{Aab} \to_{\mathcal{S}} \mathsf{Abbaa}$$

from Example 5.2. The corresponding loop in the original system is

$$\mathsf{abb} \to_{\mathcal{S}} \mathsf{bbaab} \to_{\mathcal{S}} \mathsf{bbabbaa}.$$

This is obvious since the dependency pair $\mathsf{Ab} \to \mathsf{Aa}$ employed in the first step derived from the rule $\mathsf{ab} \to \mathsf{bbaa}$.

Hence for every step involving a dependency pair $l^\sharp \to u^\sharp$ one must determine a (not necessarily unique) rule $l \to r$ from the original system that gave rise to $l^\sharp \to u^\sharp$. The additional context is then C satisfying $r = C[u]$.

Lemma 5.7. *Every sequence* $t_1^\sharp \to_{\mathsf{DP}(\mathcal{R})\cup\mathcal{R}} t_2^\sharp \to_{\mathsf{DP}(\mathcal{R})\cup\mathcal{R}} t_3^\sharp \to_{\mathsf{DP}(\mathcal{R})\cup\mathcal{R}} \cdots$ *can be transformed into a sequence* $t_1 \to_{\mathcal{R}} C_1[t_2] \to_{\mathcal{R}} C_2[t_3] \to_{\mathcal{R}} \cdots$ *involving only the original system.*

Proof. Obvious due to soundness of the dependency pair transformation [3]. □

The next result combines Theorem 5.3 and Lemma 5.7. Strings in the transformed loop might be of size larger than $n + 1$ due to the additional contexts.

Corollary 5.8. *An SRS* $\mathcal{R}$ *admits a loop of length at most* m *if the formula* $\mathit{loop}^{m+1,n}_{\mathsf{DP}(\mathcal{R}),\mathcal{R}} \wedge \varphi_{\mathit{cons}}$ *is satisfiable.* □

```
types 'a brel     = "('a × 'a)set"
types ('f,'v)loop = "('f,'v)term list × ('f,'v)ctxt × ('f,'v)sub"

definition SN_elt where
 "SN_elt 𝒜 a ≡ ¬(∃s. s₀ = a ∧ (∀i. (sᵢ,sᵢ₊₁) ∈ 𝒜))"

definition SN where "SN(𝒜) ≡ ∀a. SN_elt 𝒜 a"

fun rsteps :: "('f,'v)term list ⇒ ('f,'v)term brel ⇒ bool"
where "rsteps [t] ℛ = True"
    | "rsteps (s#t#ts) ℛ = (s →ℛ t ∧ rsteps (t#ts) ℛ)"

fun is_loop :: "('f,'v)loop ⇒ ('f,'v)term brel ⇒ bool" where
 "is_loop (t#ts,C,σ) ℛ = rsteps (t#ts@[C[tσ]]) ℛ"

fun ith :: "('f,'v)loop ⇒ nat ⇒ ('f,'v)term" where
 "ith(t#ts,C,σ)ᵢ = (if i < length(t#ts)
   then (t#ts)!i
   else C[(ith(t#ts,C,σ)_{i−length(t#ts)})σ])"
```

Figure 5.1: Basic definitions

5.2 Formalizing Loops

In this and the following section we assume some basic knowledge of the theorem prover Isabelle [69]. All lemmas and theorems within these sections have been formally proved in IsaFoR.

In contrast to finding loops, which is restricted to the setting of string rewriting, for the formalization in IsaFoR we again consider full term rewriting. What follows is a sketch on how we formalized looping reductions in Isabelle. Figure 5.1 gives an overview of the most important function definitions and types we used. (Note however that in order to increase readability, we do not strictly follow the Isabelle syntax of IsaFoR, e.g., the application of a substitution would be $t{\cdot}\sigma$ rather than $t\sigma$ in IsaFoR.) A binary relation is represented as a set of pairs over some domain. A loop is a triple consisting of a list of terms, a context, and a substitution. For a given relation $\mathcal{A}$, an element `a` is strongly normalizing (`SN_elt`) if there does not exist an infinite sequence `s` such that $\mathtt{s}_0 = \mathtt{a}$ and for all i we have $(\mathtt{s}_i, \mathtt{s}_{i+1}) \in \mathcal{A}$. Strong normalization of a relation (`SN`) is defined via the property that all elements of the domain are strongly normalizing with respect to the relation. To guarantee that our definition of `SN` is suitable we proved an easy lemma stating equivalence to the built-in Isabelle notion of well-foundedness (`wf`), i.e., $\mathtt{SN}(\mathcal{A}) = \mathtt{wf}(\mathcal{A}^{-1})$. The rewrite relation induced by a TRS $\mathcal{R}$ ($\to_{\mathcal{R}}$) is a binary relation on terms that is closed under contexts and substitutions. The latter two concepts are defined by inductive sets but details are omitted here. The function `rsteps` checks for a list of terms if between two consecutive terms there is a rewrite step. Note that this function is partial (undefined for the empty list of terms). Then a function `is_loop` can easily be defined with the help of `rsteps`.

Up to now the basic ingredients for the formalization have been introduced. What follows is an explanation on how to show that a loop indeed gives rise to non-termination. The key idea is to explicitly construct a sequence $\mathbf{s}$ that contradicts the definition of `SN_elt`, i.e., we need to define a function that takes a loop as input and generates for every i the i-th term in the non-terminating sequence defined by the loop. Hence, if $ts = [t_1, \ldots, t_n]$ loops with context C and substitution σ then there exists an infinite rewrite sequence of the following shape:

$$\begin{array}{rrrr} t_1 \to & t_2 \to \cdots \to & t_n \to & \\ C[t_1\sigma] \to & C[t_2\sigma] \to \cdots \to & C[t_n\sigma] \to & \\ C[C[t_1\sigma]\sigma] \to & C[C[t_2\sigma]\sigma] \to \cdots \to & C[C[t_n\sigma]\sigma] \to & \cdots \end{array} \tag{5.1}$$

The function `ith` is employed to return on input i the i-th element of this sequence. Note that `ith` is undefined for empty lists but this does not pose a problem since obviously a loop has to involve at least one term. If the list of terms ts loops with context C and substitution σ then between two consecutive terms in the sequence (5.1) there clearly must be a rewrite step. The reason is that the relation $\to_{\mathcal{R}}$ is closed under contexts and substitutions. However, in the theorem prover, separate lemmas are necessary stating that `ith` fulfills these properties to construct the desired sequence $\mathbf{s}$. In IsaFoR, the main task was to prove the following lemma:

Lemma 5.9. *If* `is_loop` ℓ $\mathcal{R}$ *then for all* i *we have* $\mathtt{ith}(\ell)_i \to_{\mathcal{R}} \mathtt{ith}(\ell)_{i+1}$. □

Then we obtain the main theorem for the abstract formalization:

Theorem 5.10. *If* `is_loop` ℓ $\mathcal{R}$ *then* $\to_{\mathcal{R}}$ *is not terminating.*

Proof. From `is_loop` ℓ $\mathcal{R}$ we obtain an infinite sequence $\mathbf{s}$ by defining $\mathbf{s}_i = \mathtt{ith}(\ell)_i$. This sequence satisfies $\forall i.\ \mathbf{s}_i \to_{\mathcal{R}} \mathbf{s}_{i+1}$ due to Lemma 5.9. Hence, for the first term t of the loop ℓ we obtain $\neg\mathtt{SN\_elt} \to_{\mathcal{R}} t$ and thus by definition of `SN`, $\neg\mathtt{SN}(\to_{\mathcal{R}})$. □

5.3 Certifying Loops

In contrast to the formalization above where we proved that looping reductions are non-terminating (which has been well-known for years), this section aims at certification, i.e., an automatic check if a suspected loop indeed is a loop. Nevertheless the formalization above was a necessary step for proving the check-functions to be correct. To obtain the executable check-functions we use the code-generation [36] facilities of Isabelle which allow to generate verified code for several functional programming languages. However, in our development we had to be careful, since not all Isabelle constructs admit a transformation into executable code. We provide an implementation of the predicate `is_loop` from the previous section by the check-function `check_loop` that tests if a list of terms, a context, and a substitution form a loop. Before taking a closer look at this function we state our main theorem for certifying loops:

Theorem 5.11. *If* `check_loop` ℓ $\mathcal{R}$ *then* $\to_{\mathtt{set}(\mathcal{R})}$ *is not terminating (where* `set` *transforms a list into a set).* □

```
types ('f,'v)rule = "('f,'v)term × ('f,'v)term"
types ('f,'v)trsL = "('f,'v)rule list"

fun rewrites where
 "rewrites (s,t) C σ (l,r) ℛ = (s = C[lσ] ∧ t = C[rσ] ∧ (l,r) mem ℛ)"

fun rewrites_to
where "rewrites_to [(s,C,σ,rule)] t ℛ = rewrites (s,t) C σ rule ℛ"
    | "rewrites_to ((s,C,σ,rule)#(t,C',σ',rule')#xs) u ℛ = (
       rewrites (s,t) C σ rule ℛ ∧
       rewrites_to ((t,C',σ',rule')#xs) u ℛ)"

fun check_loop_d
where "check_loop_d [] _ _ _ = False"
    | "check_loop_d ts C σ ℛ = rewrites_to ts C[(fst(hd ts))σ] ℛ"
```

Figure 5.2: Checking a loop with all details provided

This closely resembles Theorem 5.10, only on a *constructive* (meaning executable) level. The reason why $\mathcal{R}$ was chosen as a list, is that for lists executable code can be generated unlike for sets. Before actually considering the function `check_loop` we focus on a simpler task, namely a function `check_loop_d` where more details about the loop are supplied to the function, i.e., for every rewrite step $s \to_{\mathcal{R}} t$ the context C, the substitution σ, and the rewrite rule $l \to r$ such that $s = C[l\sigma] \to_{\mathcal{R}} C[r\sigma] = t$ are explicitly available. Since for this setting, no information has to be computed by the theorem prover, the implementation of `check_loop_d` is based on just the few functions depicted in Figure 5.2.

At this point certification of a candidate loop is already possible:

Lemma 5.12. *If* `check_loop_d` $xs\ C\ \sigma\ \mathcal{R}$ *then* $\to_{\texttt{set}(\mathcal{R})}$ *is not terminating.*

Proof. The abstract formalization can be linked to the concrete implementation:

If `rewrites_to` $xs\ t\ \mathcal{R}$ then `rsteps` (`map fst` xs`@`$[t]$) (`set`$(\mathcal{R})$).

By unfolding the definitions of `check_loop_d` and `is_loop`, and using Theorem 5.10, the proof concludes. □

The main drawback of the function `check_loop_d` is that it requires explicit information about the rewrite steps. To our knowledge not a single termination prover provides all these details. To make the certification of loops more appealing and user-friendly we turn our focus on the function `check_loop` again. Here for every rewrite step $s \to_{\mathcal{R}} t$ the context C, the substitution σ, and the rewrite rule $l \to r \in \mathcal{R}$ such that $s = C[l\sigma]$ and $t = C[r\sigma]$ are computed by the theorem prover.

The function `get` $s\ t\ \mathcal{R}$ computes `Some`$(C, \sigma, (l, r))$ if there is a rewrite step from s to t involving C, σ, and $l \to r \in \mathcal{R}$. Hence `get` has to test for all rules $l \to r \in \mathcal{R}$ if for any context C in s there is a substitution σ satisfying $s = C[l\sigma]$

and $t = C[r\sigma]$.[3] To find this substitution we had to implement matching. With the help of `get` a function `get_list` returns the necessary information for a sequence of rewrite steps and `get_loop` computes all details for a looping reduction. Finally the function `check_loop` just calls `check_loop_d` on the output of `get_loop`.

5.4 Experiments

This section is separated into two parts. In the first part (Section 5.4.1) we show the power of the method proposed in Section 5.1 for SRSs. The second part (Section 5.4.2) then considers more approaches for finding loops since the emphasis is put on certifying as many loops as possible.

5.4.1 Finding Loops

We integrated the encoding from Section 5.1 into T$_{\text{T}}$T$_2$. Solving the (propositional) formulas by MiniSat [20] produced slightly better results than employing Yices [18] as back-end.

The actual implementation of the encoding differs a bit from the presentation in Section 5.1 for reasons of readability. Our experiments showed that non-termination proving power can be slightly (i.e., a gain of about 10%) extended by addressing the following issues. Mutual exclusion of the M^a_{ij} variables can be expressed more concisely. After fixing an order on the variables, the property that at most one of the variables $x_1, \ldots, x_n$ can be satisfied, is expressed by $x_i \to \neg x_{i+1} \wedge \cdots \wedge \neg x_n$ for all $1 \leqslant i < n$. Due to mutual exclusion of the M^a_{ij} variables, all bi-implications occurring in subformulas of $\mathsf{loop}^{m,n}_{\mathcal{P},\mathcal{R}}$ can safely be replaced by implications. In the constraint $\mathrm{e}_{i+1} + \|l\| = \mathrm{e}_i + \|r\|$ the "=" could be weakened to "$\leqslant$". (This corresponds to cutting parts of the substitution.) However, our experiments revealed that due to the increased search space the more restrictive version performs much better. To reduce the search space our implementation employs a heuristic for p_i. Fixing $\mathrm{p}_i \leqslant \min\{2 + i * \max_{l\to r\in\mathcal{R}}\{\|l\|, \|r\|\}, n\}$ ensures that for the first few rows rewrite rules are applied close to the root. In addition to the DP processor for non-termination we employ termination methods like matrix interpretations [22] which allow to pre-process DP problems. Sometimes these single termination methods already suffice to prove termination. But more importantly, DP problems are often decreased in size before being handed over to the loop finder. The heuristic for encoding loops tries matrices of different dimensions ranging from 4×4 up to 25×25. Most successful proofs only take a few seconds.

We compare our implementation with three powerful non-termination analyzers (the latter two completely devoted to non-termination), namely Matchbox [83], nonloop [70], and NTI [71]. These tools participated in the Standard SRS categories of the 2007 or 2008 termination competitions and were (apart

[3] For TRSs the latter condition is automatically fulfilled by the requirement that all variables in r also occur in l. Since our test set (cf. Section 1.4) contains systems that violate this restriction we demand both conditions in our implementation. Consequently we can certify loops that are due to fresh variables in right-hand sides.

Table 5.1: Finding loops for 732 SRSs

	Matchbox	nonloop	NTI	T$_{\text{T}}$T$_2$
no	119	94	21	87
total time	16981	37860	35929	24051

from T$_{\text{T}}$T$_2$) the most powerful ones concerning non-termination in this division. Table 5.1 presents a comparison of the different provers (on the 732 SRSs used in the latest edition of the competition) where the row labeled no shows the number of successful non-termination proofs and the total time is measured in seconds. The strategy used for T$_{\text{T}}$T$_2$ is listed in Section A.4. When comparing the power of the tools one should not forget that the algorithm underlying NTI performs much better for terms than for strings (cf. Section 5.4.2 and [71]) and that nonloop and T$_{\text{T}}$T$_2$ use just a single non-termination method whereas Matchbox employs an enumeration of forward closures, match-bounds of inverse systems [83], reversing, and transport systems [84]. The last method is especially suitable for detecting long loops (which our approach typically misses). However, it is not (yet) possible to certify loops due to transport systems which is in contrast to the loops found by T$_{\text{T}}$T$_2$. A similar argument holds for nonloop; It claims several systems to be non-looping and non-terminating but certification of these claims is currently not possible.

5.4.2 Certifying Loops

Our contribution amounts to approximately 500 lines of Isabelle code that where added to IsaFoR (theory `Loop`). This includes the abstract formalization of loops and both approaches for certifying loops (the detailed one using `check_loop_d` and the user-friendly one based on `check_loop`). As already mentioned the key concept for the way of certification presented here, is the code-generation mechanism of Isabelle which allows to export *verified* Haskell code. Thus the whole certifier consists of a bunch of Haskell sources automatically generated by Isabelle, plus a main file that just calls the check function on a given problem and proof, i.e., when calling CeTA, two arguments have to be supplied, namely the input problem and the proof attempt. The tool then tests if the specified proof attempt corresponds to a loop and terminates with exit code 0 in case of success and exit code 1 if the input could not be proved to be a loop.

Table 5.2 contains separate empirical data for 1391 TRSs and 732 SRSs where we configured T$_{\text{T}}$T$_2$ such that it searches for loops with the method proposed in [71] (TRSs) and the approach introduced in Section 5.1 (SRSs). Furthermore two trivial methods are employed (test for fresh variables on right-hand sides and test if a rule is self-embedding). Within the table, the row T$_{\text{T}}$T$_2$ refers to finding a looping reduction and CeTA to certifying it. The columns labeled no indicate the number of successfully proved non-terminating (T$_{\text{T}}$T$_2$) and certified (CeTA) systems and total time resembles the accumulated time in seconds for finding/certifying loops. The numbers in parentheses denote the average time

Table 5.2: Certifying loops

	TRSs		SRSs	
	no	total time	no	total time
T$_T$T$_2$	212	118 (0.56)	87	1244 (14.29)
CeTA	212	9 (0.04)	87	4 (0.04)

that was needed per system for proving/certifying non-termination which shows that the computational effort for certification is negligible. CeTA could certify all 212 TRSs and 87 SRSs non-terminating for which T$_T$T$_2$ provided a loop.

Since no official release of Rainbow/CoLoR and CiME/A3PAT currently supports certification of non-termination we do not consider them for comparison with our work.

5.5 Future Work

Both contributions presented in this chapter may be further investigated. One interesting question concerning Section 5.1 is whether the encoding can be lifted from strings to terms. Concerning the formalization of loops one could try to incorporate the approach from [70] and also formalize non-looping non-termination. It has to be clarified if from the output provided by nonloop one can extract the i-th term in a non-terminating sequence easily. This issue will then make the task either easy or undoable. Other ideas concerning the formalization affect loops within the dependency pair framework. Our approach currently cannot handle DP loops but due to Lemma 5.7 this is no real restriction. Another point of interest is to certify loops under specific rewrite strategies [81].

5.6 Summary

This chapter described a new method dedicated to finding loops for SRSs. Since the encoding for loops takes parameters for the length of looping sequences and the maximal size of strings occurring within the reduction it is especially suitable to find short(est) looping evidence. This eases the task of debugging since the reason for non-termination is concisely represented. In the second part we formalized strong normalization in the theorem prover Isabelle and sketched how our contribution allows to generate verified code capable of certifying loops. The thereby generated check-function was incorporated into CeTA. This makes our work the first contribution for certifying non-termination in the field of rewriting. Experimental results show the power of our loop-finder and the efficiency of the check-functions we contributed to CeTA.

Chapter 6

Solving Arithmetic Constraints

This chapter gives some insight into the basic ideas on which the constraint solving module of TTT2 relies. Within TTT2 constraints are encoded in a language similar to the one from Definition 1.11 on page 10. Just at the time of solving the constraints it is decided which solver has to do the job. For arbitrary arithmetic constraints only the SAT back-end is suitable whereas for the linear arithmetic fragment both, SAT and SMT solvers, can be employed. In Section 6.1 we explain how to reduce arithmetic over $\mathbb{N}$ ($\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$) to SAT whereas Section 6.2 provides some pieces of information on the SMT back-end. Independent from the solver, already at the time of encoding obvious simplifications like

$$\varphi \wedge \top \to \varphi \qquad \top \wedge \varphi \to \varphi \qquad \varphi \wedge \bot \to \bot \qquad \bot \wedge \varphi \to \bot \qquad \cdots$$

are performed which help to reduce the size of the encoding. Section 6.3 stresses the main benefit of the stand-alone constraint solving module.

6.1 Transforming Arithmetic Constraints to SAT

In the case of SAT one must fix the domain of arithmetic expressions, i.e., $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$. Furthermore, to get formulas of finite size, every arithmetic variable must be bounded from above, i.e., must be represented by a given number of bits. Then operations such as $+$, $-$, $\times$, $>$, and $=$ can be unfolded according to their definitions using circuits. Such definitions have already been presented in [22] for bit-vectors of a fixed width. In contrast, we take overflows into account. The resulting constraint is propositional and can be solved by a SAT solver (after a satisfiability preserving transformation, e.g., [82, 72]).

In the sequel we start defining operations over bit-vectors (representing natural numbers) and lift them to integers, rationals, and finally (a fragment of) reals. The propositional encodings of $>_{\mathbf{N}}$ and $=_{\mathbf{N}}$ for bit-vectors given below are similar to the ones in [10] (apart from some slight optimizations).

6.1.1 Arithmetic over $\mathbb{N}$

We fix the number k of bits that is available for representing natural numbers in binary. Let $a < 2^k$. We denote by $\mathbf{a}_k = \langle a_k, \ldots, a_1 \rangle$ the binary representation of a where a_k is the most significant bit. Whenever k is not essential we abbreviate $\mathbf{a}_k$ to $\mathbf{a}$. Furthermore the operation $(\cdot)_k$ on bit-vectors is used to drop bits, i.e., $\langle a_4, a_3, a_2, a_1 \rangle_2 = \langle a_2, a_1 \rangle$.

Definition 6.1. For natural numbers given in binary representation, the operations $>_{\mathbf{N}}$ and $=_{\mathbf{N}}$ are defined as follows:

$$\begin{aligned}
\mathbf{a}_k >_{\mathbf{N}} \mathbf{b}_k &= \begin{cases} a_1 \wedge \neg b_1 & \text{if } k = 1 \\ (a_k \wedge \neg b_k) \vee \big((b_k \to a_k) \wedge \mathbf{a}_{k-1} >_{\mathbf{N}} \mathbf{b}_{k-1}\big) & \text{if } k > 1 \end{cases} \\
\mathbf{a}_k =_{\mathbf{N}} \mathbf{b}_k &= \bigwedge_{i=1}^{k} (a_i \leftrightarrow b_i)
\end{aligned}$$

For addition one has to take overflows into account. Since two k-bit integers might sum up to a $(k+1)$-bit number an additional bit is needed for the result. Consequently the case arises when two summands are not of equal bit-width. Thus, before adding $\mathbf{a}_k$ and $\mathbf{b}_{k'}$ the shorter one is padded with $|k - k'|$ $\bot$'s. To keep the presentation simple we assume that $\bot$-padding is implicitly performed before the operations $+_{\mathbf{N}}$, $>_{\mathbf{N}}$, and $=_{\mathbf{N}}$.

Definition 6.2. We define $\mathbf{a}_k +_{\mathbf{N}} \mathbf{b}_k$ as $\langle c_k, s_k, \ldots, s_1 \rangle$ for $1 \leqslant i \leqslant k$ with

$$\begin{aligned}
c_0 &= \bot \\
s_i &= a_i \oplus b_i \oplus c_{i-1} \\
c_i &= (a_i \wedge b_i) \vee (a_i \wedge c_{i-1}) \vee (b_i \wedge c_{i-1})
\end{aligned}$$

were $\oplus$ denotes exclusive or.

Note that although theoretically not necessary, in practice it is essential to introduce new variables for the carry and the sum. The reason is that in consecutive additions each bit a_i and b_i is duplicated (twice for the carry and once for the sum) and consequently using fresh variables for the sum prevents an exponential blowup of the resulting formula.

A further method to keep formulas small is to give an upper bound on the bit-width when representing naturals. This can be accomplished after addition (or multiplication) by fixing a maximal number m of bits. To restrict $\mathbf{a}_k$ to m bits we just demand that all a_i for $m + 1 \leqslant i \leqslant k$ are $\bot$ as a side constraint. Then it is safe to continue any computations with $\mathbf{a}_m$ instead of $\mathbf{a}_k$. Here safe means that restricting bits cannot produce incorrect results. However, in the worst case the side constraint can make the whole formula unsatisfiable.

The next example demonstrates addition. To ease readability we do not use arbitrary propositional variables in the following examples but just the constants $\bot$ and $\top$ and immediately perform obvious simplifications.

Example 6.3. We compute $3 +_{\mathbb{N}} 14 = 17$, i.e., $\langle \top, \top \rangle +_{\mathbf{N}} \langle \top, \top, \top, \bot \rangle = \langle \top, \bot, \bot, \bot, \top \rangle$. In the sequence below the first step extends both operands to equal bit-width. Afterwards Definition 6.2 applies.

$$\begin{aligned}
\langle \top, \top \rangle +_{\mathbf{N}} \langle \top, \top, \top, \bot \rangle &= \langle \bot, \bot, \top, \top \rangle +_{\mathbf{N}} \langle \top, \top, \top, \bot \rangle \\
&= \langle \top, \bot, \bot, \bot, \top \rangle
\end{aligned}$$

Multiplication is formulated by repeated addition and bit-shifting. The latter is denoted by $\ll$ where $\mathbf{a} \ll n$ denotes a left-shift of $\mathbf{a}$ by n bits, e.g., the expression $\langle x, y\rangle \ll 3$ yields $\langle x, y, \bot, \bot, \bot\rangle$. Another useful operation is $(\cdot)$ taking a bit-vector and a Boolean variable as input and performing a scalar multiplication, i.e., $\mathbf{a}_k \cdot x = \langle a_k \wedge x, \ldots, a_1 \wedge x\rangle$. In the sequel the operator $(\cdot)$ binds stronger than $\ll$, i.e., $\mathbf{a} \cdot x \ll 2$ is an abbreviation for $(\mathbf{a} \cdot x) \ll 2$.

In the definition below two bit-vectors with m and n bits are multiplied. Note that the result occupies at most $m + n$ bits.

Definition 6.4. For two bit-vectors $\mathbf{a}_m$ and $\mathbf{b}_n$ we define:

$$\mathbf{a}_m \times_{\mathbf{N}} \mathbf{b}_n \quad = \quad \big((\mathbf{a}_m \cdot b_1 \ll 0) +_{\mathbf{N}} \cdots +_{\mathbf{N}} (\mathbf{a}_m \cdot b_n \ll (n-1))\big)_{m+n}$$

In the following example we demonstrate multiplication.

Example 6.5. Let $\mathbf{a} = \langle \top, \bot, \top\rangle$ and $\mathbf{b} = \langle \top, \top, \bot\rangle$. Hence the example computes $5 \times_{\mathbb{N}} 6 = 30$. In the computation below the first step unfolds Definition 6.4. Then the scalar multiplications are evaluated before shifting is performed. After addition (using $+_{\mathbf{N}}$) the sum is restricted to six bits.

$$\begin{aligned}
\mathbf{a} \times_{\mathbf{N}} \mathbf{b} &= \big((\mathbf{a} \cdot \bot \ll 0) +_{\mathbf{N}} (\mathbf{a} \cdot \top \ll 1) +_{\mathbf{N}} (\mathbf{a} \cdot \top \ll 2)\big)_6 \\
&= \big((\langle \bot, \bot, \bot\rangle \ll 0) +_{\mathbf{N}} (\mathbf{a} \ll 1) +_{\mathbf{N}} (\mathbf{a} \ll 2)\big)_6 \\
&= \big(\langle \bot, \bot, \bot\rangle +_{\mathbf{N}} \langle \top, \bot, \top, \bot\rangle +_{\mathbf{N}} \langle \top, \bot, \top, \bot, \bot\rangle\big)_6 \qquad (\star) \\
&= \big(\langle \bot, \bot, \bot, \bot\rangle +_{\mathbf{N}} \langle \top, \bot, \top, \bot\rangle +_{\mathbf{N}} \langle \top, \bot, \top, \bot, \bot\rangle\big)_6 \\
&= \big(\langle \bot, \top, \bot, \top, \bot\rangle +_{\mathbf{N}} \langle \top, \bot, \top, \bot, \bot\rangle\big)_6 \\
&= \big(\langle \bot, \top, \top, \top, \top, \bot\rangle\big)_6 \\
&= \langle \bot, \top, \top, \top, \top, \bot\rangle
\end{aligned}$$

Note that it is preferable to add the summands in the row marked $(\star)$ from left to right since otherwise the (intermediate) results occupy more bits due to superfluous $\bot$-padding for addition, i.e.,

$$\begin{aligned}
\mathbf{a} \times_{\mathbf{N}} \mathbf{b} &= \cdots \\
&= \big(\langle \bot, \bot, \bot\rangle +_{\mathbf{N}} \langle \top, \bot, \top, \bot\rangle +_{\mathbf{N}} \langle \top, \bot, \top, \bot, \bot\rangle\big)_6 \\
&= \big(\langle \bot, \bot, \bot\rangle +_{\mathbf{N}} \langle \bot, \top, \bot, \top, \bot\rangle +_{\mathbf{N}} \langle \top, \bot, \top, \bot, \bot\rangle\big)_6 \\
&= \big(\langle \bot, \bot, \bot\rangle +_{\mathbf{N}} \langle \bot, \top, \top, \top, \top, \bot\rangle\big)_6 \\
&= \big(\langle \bot, \bot, \bot, \bot, \bot, \bot\rangle +_{\mathbf{N}} \langle \bot, \top, \top, \top, \top, \bot\rangle\big)_6 \\
&= \big(\langle \bot, \bot, \top, \top, \top, \top, \bot\rangle\big)_6 \\
&= \langle \bot, \top, \top, \top, \top, \bot\rangle
\end{aligned}$$

demonstrates such a case.

Before considering arithmetic over $\mathbb{Z}$ we focus on the if-then-else operator $(\cdot\ ?\ \cdot : \cdot)$ from Definition 1.11. It can be implemented using scalar multiplication and addition, i.e.,

$$x\ ?\ \mathbf{a} : \mathbf{b} \quad = \quad (\mathbf{a} \cdot x) +_{\mathbf{N}} (\mathbf{b} \cdot \neg x).$$

Hence above expression returns the value of $\mathbf{a}$ if x evaluates to true and the value of $\mathbf{b}$ otherwise. Since this operator serves just as an abbreviation we do not give its straight-forward redefinitions when considering arithmetic over $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$. Note that this operator is very useful to encode the maximum of two numbers, i.e., $\max(\mathbf{a}, \mathbf{b})$ amounts to $\mathbf{a} >_{\mathbf{N}} \mathbf{b} \; ? \; \mathbf{a} : \mathbf{b}$.

6.1.2 Arithmetic over $\mathbb{Z}$

For dealing with integers an obvious choice is to represent numbers in two's complement. The main benefit is the rather simple implementation for arithmetic operations. Here, for a k-bit number the most significant bit resembles the sign, e.g., $\mathbf{a}_k = \langle a_k, \ldots, a_1 \rangle$ with sign a_k and bits $a_{k-1}, \ldots, a_1$. As usual, a sign $\top$ indicates negative values. Again some definitions expect both operands to be of equal bit-width. This is accomplished by implicitly sign-extending the shorter operand beforehand. The operation $(\cdot)_k$ is abused for both, sign-extending and discarding bits, e.g., $\langle \bot, \top \rangle_4 = \langle \bot, \bot, \bot, \top \rangle$ and $\langle \top, \top \rangle_4 = \langle \top, \top, \top, \top \rangle$. Hence the integer represented by the bit-vector does not change when sign-extending. Similar to the case for $\mathbf{N}$ a bit-vector $\mathbf{a}_k$ can be restricted to m bits. If the dropped bits take the same value as the sign, then $\mathbf{a}_k$ and $\mathbf{a}_m$ denote the same number (this operation can be seen as the converse of sign-extending, i.e., sign-dropping). Hence if a side constraint $a_k \leftrightarrow a_i$ is added for $m \leqslant i \leqslant k$ then it is safe to proceed any computations with $\mathbf{a}_m$ instead of $\mathbf{a}_k$.

Comparisons are defined based on the corresponding operations over $\mathbf{N}$.

Definition 6.6. For two bit-vectors $\mathbf{a}_k$ and $\mathbf{b}_k$ we define:

$$\begin{aligned} \mathbf{a}_k >_{\mathbf{Z}} \mathbf{b}_k &= \big(\neg a_k \wedge b_k\big) \vee \big((a_k \to b_k) \wedge \mathbf{a}_{k-1} >_{\mathbf{N}} \mathbf{b}_{k-1}\big) \\ \mathbf{a}_k =_{\mathbf{Z}} \mathbf{b}_k &= \mathbf{a}_k =_{\mathbf{N}} \mathbf{b}_k \end{aligned}$$

The interesting case is $>_{\mathbf{Z}}$ where a separate check for the sign is needed, i.e., $\mathbf{a}$ is greater than $\mathbf{b}$ if $\mathbf{b}$ is negative while $\mathbf{a}$ is non-negative and in case of identical signs the bits are just compared using $>_{\mathbf{N}}$.

Next we define $+_{\mathbf{Z}}$ and $\times_{\mathbf{Z}}$.

Definition 6.7. Addition and multiplication are defined as follows:

$$\begin{aligned} \mathbf{a}_k +_{\mathbf{Z}} \mathbf{b}_k &= (\mathbf{a}_{k+1} +_{\mathbf{N}} \mathbf{b}_{k+1})_{k+1} \\ \mathbf{a}_m \times_{\mathbf{Z}} \mathbf{b}_n &= (\mathbf{a}_{m+n} \times_{\mathbf{N}} \mathbf{b}_{m+n})_{m+n} \end{aligned}$$

The basic idea for $+_{\mathbf{Z}}$ and $\times_{\mathbf{Z}}$ is to first sign-extend the numbers and then use the corresponding operation over the naturals. Superfluous bits are discarded afterwards.

Before demonstrating addition and multiplication by means of an example we stress that subtraction is obtained for free when using two's complement representation, i.e., $\mathbf{a} -_{\mathbf{Z}} \mathbf{b} = \mathbf{a} +_{\mathbf{Z}} \mathrm{tc}(\mathbf{b})$ where $\mathrm{tc}(\mathbf{b})$ computes two's complement of $\mathbf{b}$ which we introduce next.

Definition 6.8. For a bit-vector $\mathbf{a}_k$ we define *ones' complement* as

$$\mathrm{oc}(\mathbf{a}_k) = \langle \neg a_k, \ldots, \neg a_1 \rangle$$

and *two's complement* as

$$\mathrm{tc}(\mathbf{a}_k) = \big(\mathrm{oc}(\mathbf{a}_{k+1}) +_{\mathbf{N}} \langle\top\rangle\big)_{k+1}.$$

Basically ones' complement flips all bits and two's complement computes ones' complement incremented by one. The definition of two's complement is a bit more challenging here than usual due to bit-vectors of non-constant width. The main trick to avoid a case distinction depending on the sign is to first sign-extend the bit-vector by one auxiliary bit. After computing ones' complement, one is added and then the overflow bit is discarded. The following example demonstrates this procedure in more detail.

Example 6.9. First we show that the most significant bit must be disregarded after computing two's complement. This is important for 0, since two's complement of 0 should again return 0. We demonstrate this with 0 represented by two bits, using an additional bit for the sign. In the sequence below the first step unfolds Definition 6.8. After sign-extending oc$'s$ operand by one additional bit ones' complement is computed. The result is incremented by one and the overflow bit is discarded afterwards.

$$\begin{aligned}
\mathrm{tc}(\langle\bot,\bot,\bot\rangle) &= \big(\mathrm{oc}(\langle\bot,\bot,\bot\rangle_4) +_{\mathbf{N}} \langle\top\rangle\big)_4 \\
&= \big(\mathrm{oc}(\langle\bot,\bot,\bot,\bot\rangle) +_{\mathbf{N}} \langle\top\rangle\big)_4 \\
&= \big(\langle\top,\top,\top,\top\rangle +_{\mathbf{N}} \langle\top\rangle\big)_4 \\
&= \big(\langle\top,\bot,\bot,\bot,\bot\rangle\big)_4 \\
&= \langle\bot,\bot,\bot,\bot\rangle
\end{aligned}$$

Next we show that sign-extending is important before performing ones' complement. To this end we calculate two's complement of -8 which correctly evaluates to 8.

$$\begin{aligned}
\mathrm{tc}(\langle\top,\bot,\bot,\bot\rangle) &= \big(\mathrm{oc}(\langle\top,\bot,\bot,\bot\rangle_5) +_{\mathbf{N}} \langle\top\rangle\big)_5 \\
&= \big(\mathrm{oc}(\langle\top,\top,\bot,\bot,\bot\rangle) +_{\mathbf{N}} \langle\top\rangle\big)_5 \\
&= \big(\langle\bot,\bot,\top,\top,\top\rangle +_{\mathbf{N}} \langle\top\rangle\big)_5 \\
&= \big(\langle\bot,\bot,\top,\bot,\bot,\bot\rangle\big)_5 \\
&= \langle\bot,\top,\bot,\bot,\bot\rangle
\end{aligned}$$

The next example focuses on addition/subtraction and multiplication.

Example 6.10. We compute $5 -_{\mathbb{Z}} 2 = 3$, i.e., $\langle\bot,\top,\bot,\top\rangle -_{\mathbf{Z}} \langle\bot,\top,\bot\rangle = \langle\bot,\bot,\bot,\top,\top\rangle$. The sequence below translates subtraction ($-_{\mathbf{Z}}$) into addition ($+_{\mathbf{Z}}$) in the first step. Then two's complement of 2 is calculated. Afterwards addition for integers is performed by first sign-extending both operands by one additional bit and then performing addition for naturals ($+_{\mathbf{N}}$). After this step

the superfluous carry bit is disregarded, i.e.,

$$\begin{aligned}\langle\bot,\top,\bot,\top\rangle -_{\mathbf{Z}} \langle\bot,\top,\bot\rangle &= \langle\bot,\top,\bot,\top\rangle +_{\mathbf{Z}} \mathrm{tc}(\langle\bot,\top,\bot\rangle)\\ &= \langle\bot,\top,\bot,\top\rangle +_{\mathbf{Z}} \langle\top,\top,\top,\bot\rangle\\ &= \big(\langle\bot,\top,\bot,\top\rangle_5 +_{\mathbf{N}} \langle\top,\top,\top,\bot\rangle_5\big)_5\\ &= \big(\langle\bot,\bot,\top,\bot,\top\rangle +_{\mathbf{N}} \langle\top,\top,\top,\top,\bot\rangle\big)_5\\ &= \big(\langle\top,\bot,\bot,\bot,\top,\top\rangle\big)_5\\ &= \langle\bot,\bot,\bot,\top,\top\rangle.\end{aligned}$$

Multiplication is similar, i.e., both operands $\mathbf{a}_m$ and $\mathbf{b}_n$ are first sign-extended to have $m+n$ bits. After the multiplication ($\times_{\mathbf{N}}$) only the relevant $m+n$ bits are taken. We demonstrate multiplication by computing $5 \times_{\mathbb{Z}} -2 = -10$, i.e., $\langle\bot,\top,\bot,\top\rangle \times_{\mathbf{Z}} \langle\top,\top,\bot\rangle = \langle\top,\top,\top,\bot,\top,\top,\bot\rangle$:

$$\begin{aligned}&\langle\bot,\top,\bot,\top\rangle \times_{\mathbf{Z}} \langle\top,\top,\bot\rangle\\ &\quad= \big(\langle\bot,\top,\bot,\top\rangle_7 \times_{\mathbf{N}} \langle\top,\top,\bot\rangle_7\big)_7\\ &\quad= \big(\langle\bot,\bot,\bot,\bot,\top,\bot,\top\rangle \times_{\mathbf{N}} \langle\top,\top,\top,\top,\top,\top,\bot\rangle\big)_7\\ &\quad= \big(\langle\bot,\bot,\bot,\bot,\top,\bot,\bot,\top,\top,\top,\bot,\top,\top,\bot\rangle\big)_7\\ &\quad= \langle\top,\top,\top,\bot,\top,\top,\bot\rangle\end{aligned}$$

6.1.3 Arithmetic over $\mathbb{Q}$

Rationals are represented as a pair of numerator and denominator. The numerator is a bit-vector representing an integer whereas the denominator is a positive integer (negative denominators would demand a case analysis for $>_{\mathbf{Q}}$). All operations with the exception of $\times_{\mathbf{Q}}$ require identical denominators. This can easily be established by expanding the fractions beforehand (as demonstrated in Example 6.13).

Comparisons using rationals are just like for integers if the denominators of the two operands coincide.

Definition 6.11. For $(\mathbf{a}, d)$ and $(\mathbf{b}, d)$ representing rationals we define:

$$\begin{aligned}(\mathbf{a},d) >_{\mathbf{Q}} (\mathbf{b},d) &= \mathbf{a} >_{\mathbf{Z}} \mathbf{b}\\ (\mathbf{a},d) =_{\mathbf{Q}} (\mathbf{b},d) &= \mathbf{a} =_{\mathbf{Z}} \mathbf{b}\end{aligned}$$

The operations $+_{\mathbf{Q}}$ and $\times_{\mathbf{Q}}$ are inspired from the corresponding operations over fractions.

Definition 6.12. Addition and multiplication are defined as follows:

$$\begin{aligned}(\mathbf{a},d) +_{\mathbf{Q}} (\mathbf{b},d) &= (\mathbf{a} +_{\mathbf{Z}} \mathbf{b}, d)\\ (\mathbf{a},d) \times_{\mathbf{Q}} (\mathbf{b},d') &= (\mathbf{a} \times_{\mathbf{Z}} \mathbf{b}, d \times d')\end{aligned}$$

The next example demonstrates addition for rational numbers. Multiplication is easier since the operands' denominators need not be equal and is thus not considered in the example.

Example 6.13. Consider $\frac{3}{2} +_{\mathbb{Q}} \frac{-1}{4} = \frac{5}{4}$, i.e., $(\langle\bot,\top,\top\rangle,2) +_{\mathbf{Q}} (\langle\top,\top\rangle,4) = (\langle\bot,\bot,\top,\bot,\top\rangle,4)$. In the computation below, first both denominators are made equal. Afterwards addition of the numerators is performed using $+_{\mathbf{Z}}$:

$$\begin{aligned}(\langle\bot,\top,\top\rangle,2) +_{\mathbf{Q}} (\langle\top,\top\rangle,4) &= (\langle\bot,\top,\top,\bot\rangle,4) +_{\mathbf{Q}} (\langle\top,\top\rangle,4)\\ &= (\langle\bot,\top,\top,\bot\rangle +_{\mathbf{Z}} \langle\top,\top\rangle,4)\\ &= (\langle\bot,\bot,\top,\bot,\top\rangle,4)\end{aligned}$$

6.1.4 Arithmetic over $\mathbb{R}$

It turned out that arithmetic over $\mathbb{R}$ is most challenging. First of all in contrast to $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$ it is not so clear how numbers from $\mathbb{R}$ can be represented using bit-vectors. To allow a finite representation we only deal with a subset of $\mathbb{R}$ using a pair $(\mathbf{c},\mathbf{d})$ where $\mathbf{c}$ and $\mathbf{d}$ denote numbers from $\mathbf{Q}$. Such a pair $(\mathbf{c},\mathbf{d})$ has the intended semantics of $\mathbf{c}+\mathbf{d}\sqrt{2}$. But further problems arise when comparing two such abstract numbers. Therefore the definition of $>_{\mathbf{R}}$ given below is just an approximation of the actual order $>_{\mathbb{R}}$. The idea is to under-approximate $\mathbf{d}\sqrt{2}$ on the left-hand side while over-approximating it on the right-hand side since the value of $\sqrt{2}$ cannot be finitely represented as a bit-vector. We under-approximate $\mathbf{d}\sqrt{2}$ by $(\mathbf{5},4) \times_{\mathbf{Q}} \mathbf{d}$ if $\mathbf{d}$ is not negative and similarly $\mathbf{d}\sqrt{2}$ by $(\mathbf{3},2) \times_{\mathbf{Q}} \mathbf{d}$ if $\mathbf{d}$ is negative.[1] Note that $\frac{5}{4} = 1.25 <_{\mathbb{R}} 1.41 \approx \sqrt{2}$ and $-\frac{3}{2} = -1.5 <_{\mathbb{R}} -1.41 \approx -\sqrt{2}$ which justifies the approach. An analogous reasoning yields the over-approximation. This trick allows to implement $>_{\mathbf{R}}$ (an approximation of $>_{\mathbb{R}}$) based on $>_{\mathbf{Q}}$. The advantage is that $>_{\mathbf{Q}}$ can be expressed in SAT (cf. Definition 6.11).

Next we formally define the under- and over-approximation for abstract numbers $\mathbf{a}$ from $\mathbf{Q}$ depending on their sign (denoted $\mathrm{sign}(\mathbf{a})$ with obvious definition) using the if-then-else operator. Recall that a sign $\top$ indicates negative numbers.

Definition 6.14. For a number $\mathbf{a}$ from $\mathbf{Q}$ we define:

$$\begin{aligned}\mathrm{under}(\mathbf{a}) &= (\mathrm{sign}(\mathbf{a})\ ?\ (\mathbf{3},2) : (\mathbf{5},4)) \times_{\mathbf{Q}} \mathbf{a}\\ \mathrm{over}(\mathbf{a}) &= (\mathrm{sign}(\mathbf{a})\ ?\ (\mathbf{5},4) : (\mathbf{3},2)) \times_{\mathbf{Q}} \mathbf{a}\end{aligned}$$

With the over- and under-approximations we are now ready to define the comparisons $>_{\mathbf{R}}$ and $=_{\mathbf{R}}$.

Definition 6.15. For the pairs $(\mathbf{c},\mathbf{d})$ and $(\mathbf{e},\mathbf{f})$ we define:

$$\begin{aligned}(\mathbf{c},\mathbf{d}) >_{\mathbf{R}} (\mathbf{e},\mathbf{f}) &= \mathbf{c} +_{\mathbf{Q}} \mathrm{under}(\mathbf{d}) >_{\mathbf{Q}} \mathbf{e} +_{\mathbf{Q}} \mathrm{over}(\mathbf{f})\\ (\mathbf{c},\mathbf{d}) =_{\mathbf{R}} (\mathbf{e},\mathbf{f}) &= \mathbf{c} =_{\mathbf{Q}} \mathbf{e} \ \wedge\ \mathbf{d} =_{\mathbf{Q}} \mathbf{f}\end{aligned}$$

To increase readability, in the following examples we unravel the pair notation $(\mathbf{c},\mathbf{d})$ using the intended semantics to $c +_{\mathbb{R}} d\sqrt{2}$ whenever convenient. Hence $(\mathbf{3},\mathbf{1})$ amounts to $3 +_{\mathbb{R}} \sqrt{2}$.

[1] Here and in the sequel we abbreviate numbers from $\mathbf{Z}$ and $\mathbf{Q}$ by denoting them in boldface, i.e., $\mathbf{5}$ represents the number $\langle\bot,\top,\bot,\top\rangle$ from $\mathbf{Z}$ and at the same time the number $(\langle\bot,\top,\bot,\top\rangle,1)$ from $\mathbf{Q}$. The context clarifies which one is meant.

Example 6.16. The expression $(\mathbf{1},\mathbf{1}) >_{\mathbf{R}} (\mathbf{2},\mathbf{0})$ approximates the comparison $1+\sqrt{2} >_{\mathbb{R}} 2$. Next $\sqrt{2}$ on the left-hand side is under-approximated by $\frac{5}{4}$. This allows us to replace $>_{\mathbb{R}}$ by $>_{\mathbb{Q}}$. Hence $1+\frac{5}{4} >_{\mathbb{Q}} 2$, i.e., $\frac{9}{4} >_{\mathbb{Q}} \frac{8}{4}$ shows that the above comparison is valid. Note that $(\mathbf{0},\mathbf{6}) >_{\mathbf{R}} (\mathbf{0},\mathbf{5})$ does not hold. The expression resembles $6\sqrt{2} >_{\mathbb{R}} 5\sqrt{2}$. Approximating $\sqrt{2}$ as described above yields $6\times\frac{5}{4} >_{\mathbb{Q}} 5\times\frac{3}{2}$, i.e., $\frac{30}{4} >_{\mathbb{Q}} \frac{15}{2}$ which does not hold and thus demonstrates that the order is just an approximation.

The ideas for $+_{\mathbf{R}}$ and $\times_{\mathbf{R}}$ are directly inspired from the semantics of pairs.

Definition 6.17. We define addition and multiplication for pairs $(\mathbf{c},\mathbf{d})$ and $(\mathbf{e},\mathbf{f})$ as:

$$\begin{aligned}(\mathbf{c},\mathbf{d}) +_{\mathbf{R}} (\mathbf{e},\mathbf{f}) &= (\mathbf{c} +_{\mathbf{Q}} \mathbf{e}, \mathbf{d} +_{\mathbf{Q}} \mathbf{f})\\ (\mathbf{c},\mathbf{d}) \times_{\mathbf{R}} (\mathbf{e},\mathbf{f}) &= (\mathbf{c} \times_{\mathbf{Q}} \mathbf{e} +_{\mathbf{Q}} \mathbf{2} \times_{\mathbf{Q}} \mathbf{d} \times_{\mathbf{Q}} \mathbf{f}, \mathbf{c} \times_{\mathbf{Q}} \mathbf{f} +_{\mathbf{Q}} \mathbf{d} \times_{\mathbf{Q}} \mathbf{e})\end{aligned}$$

The next example demonstrates addition and multiplication for reals.

Example 6.18. The equality $(\mathbf{1},\mathbf{2}) +_{\mathbf{R}} (\mathbf{5},\mathbf{3}) = (\mathbf{6},\mathbf{5})$ is justified since the left-hand side amounts to $1 +_{\mathbb{R}} 2\sqrt{2} +_{\mathbb{R}} 5 +_{\mathbb{R}} 3\sqrt{2}$ which simplifies to $6 +_{\mathbb{R}} 5\sqrt{2}$ corresponding to the right-hand side. The product $(\mathbf{1},\mathbf{2}) \times_{\mathbf{R}} (\mathbf{5},\mathbf{3}) = (\mathbf{17},\mathbf{13})$ is similarly justified: $(1 +_{\mathbb{R}} 2\sqrt{2}) \times_{\mathbb{R}} (5 +_{\mathbb{R}} 3\sqrt{2}) = 5 +_{\mathbb{R}} 10\sqrt{2} +_{\mathbb{R}} 3\sqrt{2} +_{\mathbb{R}} 6\sqrt{2}\sqrt{2} = 17 +_{\mathbb{R}} 13\sqrt{2}$.

6.2 Transforming Arithmetic Constraints to SMT

The benefit of SMT solvers is that due to the richer input format they provide built-in support for arithmetic. Although current SMT solvers are equipped with highly efficient solving mechanisms for *linear* arithmetic they lack any useful support for non-linear arithmetic. Hence for the operator $\times$ from Definition 1.11 it is required that at least one of the operands is a concrete number. Recently [8] showed how SMT solvers that just support linear arithmetic can be used to also handle non-linear constraints. The key idea there is to restrict every arithmetic variable to a finite domain and then linearize the constraints by an explicit case analysis. A (slight) disadvantage of the SMT back-end is the lack of support for real numbers. Actually all SMT solvers currently restrict themselves to rationals and also the approach from [8] does not give any information on how to employ SMT solvers for arithmetic over the real numbers.

6.3 Constraint Solving Module

T$_{\text{T}}$T$_2$'s constraint solving module is addressed in this section. One nice aspect is the stand-alone design which also allows other applications (and other termination tools) to use it. However, currently only an interface to OCaml is provided. Furthermore, when encoding a (non-)termination method the intermediate format from Definition 1.11 is used. Hence just before solving the constraints the user has to specify the desired back-end, i.e., SAT, SMT, or PB. (We admit that PB is suitable for constraints of a very special shape only.)

To accomplish this, arithmetic variables keep information on their type (**N**, **Z**, **Q**, or **R**) and an upper bound they may take (i.e., for SAT it is essential how many bits are used to represent them). Internally all numbers have type **R**. To represent rationals in SAT the non-real part of a number is fixed to zero, e.g., $(\mathbf{c}, \mathbf{0})$ (representing $c +_{\mathbb{R}} 0\sqrt{2}$) obviously denotes the rational number **c**. Integers are rationals with fixed denominator one and naturals are integers where the sign bit is the constant $\bot$. The slight computational overhead caused by the incremental design is negligible. Furthermore this drawback is more than compensated by the gains in flexibility, i.e., this design allowed us to switch from matrices over the naturals to matrices over the rationals/reals without any adjustment of the encoding.

This chapter is concluded with some comments on solving arithmetic constraints. Although Tarski [78] showed that the first-order theory over $\mathbb{R}$ is decidable, the underlying decision procedure is not very suitable for implementation. Since for termination analysis usually rather small domains suffice we anticipate that the SAT approach outperforms Tarski's method for this special application.

Chapter 7

SAT via Termination

Encoding termination in SAT currently is a very popular and competitive research topic. As already discussed for many traditional termination criteria SAT allows concise and easy implementations beating dedicated algorithms. This is especially surprising for KBO (cf. Chapter 2) since KBO orientability is known to be decidable in polynomial time [53] whereas SAT is NP-complete [13]. In other words, reducing KBO to SAT and applying the sophisticated algorithms for solving the computationally harder (unless P = NP) problem SAT outperform the dedicated methods for KBO [17, 53]. This chapter will enlighten the question whether a similar result also holds when translating the NP-complete SAT problem into the undecidable termination property of TRSs. However, the experiments reveal that at least for our translations the results are as expected. Concerning the transformation from SAT to termination, the dedicated SAT approaches perform much better. Even further, only the most simple propositional formulas produce TRSs which can be shown terminating by state-of-the-art termination provers. Therefore the translations can be used to generate a large set of difficult termination problems automatically.

The rest of this chapter is organized as follows: In Section 7.1 propositional formulas are introduced and many-sorted rewriting is defined. In Section 7.2 we define TRSs $\mathcal{U}^\varphi$ that are terminating if and only if φ is unsatisfiable. In Section 7.3 the dual problem is considered for many-sorted TRSs $\mathcal{S}^\varphi$ and $\mathcal{T}^\varphi$ that are terminating if and only if φ is satisfiable. That even simple propositional formulas produce TRSs where termination analysis is challenging is demonstrated in Section 7.4 where it also becomes apparent that narrowing [31] is one method which can handle small instances.

The contribution of this chapter appeared in [94].

7.1 Preliminaries

In this section we fix basic notation concerning propositional logic, introduce many-sorted TRSs and define the model variant of semantic labeling [96] in a many-sorted setting. Aoto and Yamada [2] already generalized semantic labeling to many-sorted rewriting but just for the quasi-model case.

7.1.1 Propositional Logic

Let $\mathcal{A}$ be a set of propositional variables (atoms). Sometimes we find it convenient to abbreviate the set of atoms $p_1, \ldots, p_n$ by $\mathcal{A}_n$. The set of propositional

formulas $\mathcal{P}(\mathcal{A})$ is inductively defined by the following BNF:

$$\varphi ::= p \in \mathcal{A} \mid (\varphi \wedge \varphi) \mid (\neg\varphi)$$

Note that we do not allow disjunction (which does not pose a restriction but allows to keep the presentation concise). The following convention is used to reduce the number of parentheses: (i) Outermost parenthesis are omitted, (ii) '$\wedge$' is left-associative, and (iii) '$\neg$' binds stronger than '$\wedge$'.

Let $\mathbb{B} := \{0, 1\}$. An *assignment* is a mapping $\alpha\colon \mathcal{A} \to \mathbb{B}$. It is lifted to an interpretation of formulas: $[\alpha]\colon \mathcal{P}(\mathcal{A}) \to \mathbb{B}$ with

$$[\alpha](\varphi) = \begin{cases} \alpha(p) & \text{if } \varphi = p \text{ for some } p \in \mathcal{A}, \\ [\alpha](\psi) \cdot\ [\alpha](\chi) & \text{if } \varphi = \psi \wedge \chi, \\ \overline{[\alpha](\psi)} & \text{if } \varphi = \neg\psi. \end{cases}$$

Here $(\cdot)\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ is defined as $x \cdot y = 1$ if and only if $x = y = 1$ and for $\overline{(*)}\colon \mathbb{B} \to \mathbb{B}$ we have $\overline{x} = 1$ if and only if $x = 0$. A formula φ is *satisfiable (unsatisfiable)* if an (no) assignment α exists such that $[\alpha](\varphi) = 1$. This problem is known as the satisfiability problem (SAT). For a propositional formula φ its *depth* is defined as follows: $\mathrm{depth}(p) = 0$ for $p \in \mathcal{A}$, $\mathrm{depth}(\varphi \wedge \psi) = 1 + \mathsf{max}(\mathrm{depth}(\varphi), \mathrm{depth}(\psi))$, and $\mathrm{depth}(\neg\varphi) = 1 + \mathrm{depth}(\varphi)$. Similarly the set of variables $\mathcal{V}\mathsf{ar}(\varphi)$ is defined recursively by: $\mathcal{V}\mathsf{ar}(p) = \{p\}$ for $p \in \mathcal{A}$, $\mathcal{V}\mathsf{ar}(\varphi \wedge \psi) = \mathcal{V}\mathsf{ar}(\varphi) \cup \mathcal{V}\mathsf{ar}(\psi)$, and $\mathcal{V}\mathsf{ar}(\neg\varphi) = \mathcal{V}\mathsf{ar}(\varphi)$. The well-known coincidence lemma states that when testing φ for satisfiability only the (finitely many) variables that actually occur in φ have to be considered which makes SAT decidable because the search space becomes finite. Furthermore, this allows us to relate assignments to substitutions (whose domain must be finite by definition) in the next section. Despite the fact that the search space for a satisfying assignment is finite, deciding SAT is difficult, more precisely, SAT is an NP-complete problem [13].

7.1.2 Many-Sorted Semantic Labeling

Let $\mathcal{S}$ be a non-empty set of *sorts*. An *$\mathcal{S}$-sorted* signature is a set of function symbols $\mathcal{F}$, where each $f \in \mathcal{F}$ of arity n is associated with the *function signature* $\mathsf{sig}\colon \mathcal{F} \to \mathcal{S}^{n+1}$. Here the first n components of $\mathsf{sig}(f)$ give the sort (type) of each argument and the last gives the sort of the function's result. In the following, we write $f : s_1 \times \cdots \times s_n \to s_{n+1}$, to express that f has (function) signature $(s_1, \ldots, s_{n+1})$.

An $\mathcal{S}$-sorted set A is a family of sets $\{A_s\}_{s\in\mathcal{S}}$. For an $\mathcal{S}$-sorted set $\mathcal{V}$ of *variables* (where $\mathcal{V}_s \cap \mathcal{V}_t = \varnothing$ for $s \neq t$), let $\mathcal{T}(\mathcal{F}, \mathcal{V})_s$ denote the set of terms with sort s over $\mathcal{F}$ and $\mathcal{V}$, which is defined inductively by the rules

$$\frac{x \in \mathcal{V}_s}{x} \qquad \frac{f \in \mathcal{F} \quad f : s_1 \times \cdots \times s_n \to s \quad t_i \in \mathcal{T}(\mathcal{F}, \mathcal{V})_{s_i}}{f(t_1, \ldots, t_n)}.$$

This yields the $\mathcal{S}$-sorted set $\mathcal{T}(\mathcal{F}, \mathcal{V}) = \{\mathcal{T}(\mathcal{F}, \mathcal{V})_s\}_{s\in\mathcal{S}}$. Associated with every term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ is its *sort*, i.e., if $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})_s$ then $\mathsf{sort}(t) = s$. An $\mathcal{S}$-sorted

TRS $\mathcal{R}$ is an $\mathcal{S}$-sorted set of pairs $(l, r) \in \mathcal{R}_s$—the so called *rewrite rules*—written as $l \to r$, such that there exists an $s \in \mathcal{S}$ with $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{V})_s$ and the usual restrictions that l is not a variable and all variables in r also occur in l are satisfied. In the sequel we identify one-sorted TRSs with unsorted ones and feel free to omit sort information where it is not essential.

Let $\mathcal{F}$ be an $\mathcal{S}$-sorted signature. An $\mathcal{S}$-sorted $\mathcal{F}$-algebra $\mathcal{A}$ consists of an $\mathcal{S}$-sorted *carrier* A (where each $A_s \in A$ is non-empty) and a set of *interpretations* $\{f_\mathcal{A}\}_{f \in \mathcal{F}}$, such that for each function symbol $f : s_1 \times \cdots \times s_n \to s_{n+1}$ there is an interpretation $f_\mathcal{A} \colon A_{s_1} \times \cdots \times A_{s_n} \to A_{s_{n+1}}$. An $\mathcal{S}$-sorted *substitution* $\sigma \colon \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ is a set of mappings $\sigma_s \colon \mathcal{V}_s \to \mathcal{T}(\mathcal{F}, \mathcal{V})_s$ for every $s \in \mathcal{S}$ such that $\sigma(x) \neq x$ only for finitely many $x \in \mathcal{V}$. An $\mathcal{S}$-sorted *assignment* $\alpha \colon \mathcal{V} \to A$ is a set of mappings $\alpha_s \colon \mathcal{V}_s \to A_s$ for every $s \in \mathcal{S}$. For every $\mathcal{S}$-sorted term t and assignment $\alpha \colon \mathcal{V} \to A$, a mapping $[\alpha]_\mathcal{A} \colon \mathcal{T}(\mathcal{F}, \mathcal{V}) \to A$ is defined inductively

$$[\alpha]_\mathcal{A}(t) = \begin{cases} \alpha_s(x) & \text{if } t = x \text{ and } \mathsf{sort}(t) = s, \\ f_\mathcal{A}([\alpha]_\mathcal{A}(t_1), \ldots, [\alpha]_\mathcal{A}(t_n)) & \text{if } t = f(t_1, \ldots, t_n). \end{cases}$$

An $\mathcal{S}$-sorted $\mathcal{F}$-algebra is a *model* of an $\mathcal{S}$-sorted TRS, if for all $\mathcal{S}$-sorted assignments α and rewrite rules $l \to r \in \mathcal{R}$ it holds that $[\alpha]_\mathcal{A}(l) = [\alpha]_\mathcal{A}(r)$. A *labeling* L chooses for every $f \in \mathcal{F}$ a set of *labels* L_f. The *labeled signature* is defined by

$$\mathcal{F}_\mathsf{lab} = \{f \mid f \in \mathcal{F}, L_f = \varnothing\} \cup \{f_a \mid f \in \mathcal{F}, a \in L_f\}$$

where the arity and function signature of f_a and f coincide. A *labeling* ℓ for an $\mathcal{S}$-sorted algebra $\mathcal{A}$ consists of a labeling L together with a *labeling function* $\ell_f \colon A_{s_1} \times \cdots \times A_{s_n} \to L_f$ for every $f \in \mathcal{F}$ with $L_f \neq \varnothing$ and $\mathsf{sig}(f) = s_1 \times \cdots \times s_n \to s_{n+1}$. Let $A^\mathcal{V}$ denote the set of all $\mathcal{S}$-sorted assignments from $\mathcal{V}$ to A. Let ℓ be a labeling for $\mathcal{A}$. For every assignment $\alpha \in A^\mathcal{V}$ a mapping $\mathsf{lab}_\alpha \colon \mathcal{T}(\mathcal{F}, \mathcal{V}) \to \mathcal{T}(\mathcal{F}_\mathsf{lab}, \mathcal{V})$ is defined inductively as follows

$$\mathsf{lab}_\alpha(t) = \begin{cases} x & \text{if } t = x, \\ f(\mathsf{lab}_\alpha(t_1), \ldots, \mathsf{lab}_\alpha(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } L_f = \varnothing, \\ f_a(\mathsf{lab}_\alpha(t_1), \ldots, \mathsf{lab}_\alpha(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } L_f \neq \varnothing \end{cases}$$

with $a = \ell_f([\alpha]_\mathcal{A}(t_1), \ldots, [\alpha]_\mathcal{A}(t_n))$. For any $\mathcal{S}$-sorted TRS $\mathcal{R}$ over $\mathcal{F}$, together with an $\mathcal{F}$-algebra $\mathcal{A}$ and a labeling ℓ, the $\mathcal{S}$-sorted TRS $\mathcal{R}_\mathsf{lab}$ over $\mathcal{F}_\mathsf{lab}$ is given by

$$\mathcal{R}_\mathsf{lab} = \{\mathsf{lab}_\alpha(l) \to \mathsf{lab}_\alpha(r) \mid l \to r \in \mathcal{R}, \alpha \in A^\mathcal{V}\}.$$

An $\mathcal{S}$-sorted TRS $\mathcal{R}$ is *terminating* if it does not admit an infinite rewrite sequence $t_1 \to_\mathcal{R} t_2 \to_\mathcal{R} \cdots$ starting at some $t_1 \in \mathcal{T}(\mathcal{F}, \mathcal{V})_s$ for some $s \in \mathcal{S}$.

Theorem 7.1. *Let $\mathcal{R}$ be an $\mathcal{S}$-sorted TRS. Let the algebra $\mathcal{A}$ be a model of $\mathcal{R}$ and let ℓ be a labeling for $\mathcal{A}$. Then $\mathcal{R}$ is terminating if and only if $\mathcal{R}_\mathsf{lab}$ is terminating.* □

For the TRSs we are dealing with in the subsequent sections, many-sorted termination is equivalent to the one-sorted case since the systems are non-collapsing [95]. A TRS is collapsing if it contains a rule $l \to x$ for some variable x. Restricting to many-sortedness simplifies the proofs of Theorems 7.6 and 7.10 considerably.

7.2 Transforming Unsatisfiability to Termination

In the following we want to express SAT as a termination problem in rewriting, i.e., given a formula φ, we construct a TRS $\mathcal{R}^\varphi$ that is terminating if and only if φ is satisfiable. This transformation is addressed in the next section. First we focus on the simpler dual problem, namely the construction of a TRS $\mathcal{R}^\varphi$ that is terminating if and only if φ is unsatisfiable.

For this purpose we consider a $\{\mathsf{bool}\}$-sorted signature $\mathcal{F} = \{\star, -\}$ containing a binary function symbol $(\star) : \mathsf{bool} \times \mathsf{bool} \to \mathsf{bool}$ and a unary function symbol $- : \mathsf{bool} \to \mathsf{bool}$. Furthermore we assume that the propositional atoms in $\mathcal{A}$ are contained in the set of term variables $\mathcal{V}_{\mathsf{bool}}$. Although '$\star$' will represent (on the term level) the same as '$\wedge$' does on formulas, we use different function symbols because we want to clearly separate between the two different concepts. The same holds for the symbols '$-$' and '$\neg$'. The obvious encoding $\ulcorner * \urcorner \colon \mathcal{P}(\mathcal{A}) \to \mathcal{T}(\{\star, -\}, \mathcal{V})$ transforms formulas into terms as follows: $\ulcorner p \urcorner = p$ for $p \in \mathcal{A}$, $\ulcorner \varphi \wedge \psi \urcorner = \ulcorner \varphi \urcorner \star \ulcorner \psi \urcorner$, and $\ulcorner \neg \varphi \urcorner = - \ulcorner \varphi \urcorner$. Now every well-formed formula in $\mathcal{P}(\mathcal{A})$ has a corresponding term representation in $\mathcal{T}(\mathcal{F}, \mathcal{V})$.

The next goal is to mimic the task of assignments for formulas on the term level. Thus the signature $\mathcal{F}$ is extended by two constant symbols of sort bool, namely '$\bot$' and '$\top$'. We say that an assignment $\alpha \colon \mathcal{A}_n \to \mathbb{B}$ and a substitution $\sigma \colon \mathcal{V}_n \to \{\bot, \top\}$ are *corresponding* if $\alpha(p_i) = 0$ if and only if $\sigma(p_i) = \bot$ for all $1 \leqslant i \leqslant n$ (here $\mathcal{V}_n = \mathcal{A}_n$).

In order to perform the work $[\alpha]$ does on formulas the six rewrite rules

$$\bot \star \bot \to \bot \quad \bot \star \top \to \bot \quad \top \star \bot \to \bot \quad \top \star \top \to \top \quad - \bot \to \top \quad - \top \to \bot$$

referred to as the TRS $\mathcal{S}\mathsf{imp}$ are employed. The next lemma formalizes the interplay of assignments and substitutions.

Lemma 7.2. *Let $\varphi \in \mathcal{P}(\mathcal{A}_n)$ and $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that $\ulcorner \varphi \urcorner = t$. If the assignment α and the substitution σ are corresponding, then*

- *$[\alpha](\varphi) = 0$ implies $t\sigma \to^*_{\mathcal{S}\mathsf{imp}} \bot$ and dually*
- *$[\alpha](\varphi) = 1$ implies $t\sigma \to^*_{\mathcal{S}\mathsf{imp}} \top$*

Proof. By induction on the structure of φ. □

The following example already contains the main idea for constructing non-terminating sequences.

Example 7.3. Consider the formula $\varphi = p_1 \wedge \neg p_2$ with the corresponding term $t = \ulcorner \varphi \urcorner = p_1 \star (- p_2)$. Then the TRS $\mathcal{S}\mathsf{imp}$ together with the rewriting rule

$$\mathsf{unsat}(p_1, p_2, \top) \to \mathsf{unsat}(p_1, p_2, p_1 \star (- p_2))$$

admits the cyclic reduction

$$t = \mathsf{unsat}(\top, \bot, \top) \to \mathsf{unsat}(\top, \bot, \top \star (- \bot)) \to \mathsf{unsat}(\top, \bot, \top \star \top) \to t$$

which proves non-termination of this TRS. The reason for non-termination is that for a satisfying assignment α (in this case $\alpha(p_1) = 1$ and $\alpha(p_2) = 0$) there is a *corresponding* substitution σ such that the term $\ulcorner\varphi\urcorner\sigma = t\sigma$ rewrites to $\top$ by Lemma 7.2.

The next theorem formally establishes the relation between satisfiable formulas and non-termination of corresponding TRSs.

Theorem 7.4. *Let $\varphi \in \mathcal{P}(\mathcal{A}_n)$. The generic TRS $\mathcal{U}^\varphi$ that consists of all rules in $\mathcal{S}\mathsf{imp}$ and*

$$\mathsf{unsat}(p_1, \ldots, p_n, \top) \to \mathsf{unsat}(p_1, \ldots, p_n, \ulcorner\varphi\urcorner) \tag{7.1}$$

is terminating if and only if φ is unsatisfiable.

Proof. For the direction from left to right assume $\mathcal{U}^\varphi$ to be terminating and φ to be satisfiable to arrive at a contradiction. Since φ is satisfiable there must be a satisfying assignment α and a corresponding substitution σ. But then there is the cyclic reduction

$$t = \mathsf{unsat}(\sigma(p_1), \ldots, \sigma(p_n), \top) \to \mathsf{unsat}(\sigma(p_1), \ldots, \sigma(p_n), \ulcorner\varphi\urcorner\sigma) \to^* t$$

where the first rewrite step is an application of rule 7.1 and the rest of the sequence holds by Lemma 7.2 since $\ulcorner\varphi\urcorner\sigma \to^*_{\mathcal{S}\mathsf{imp}} \top$. Contradiction.

For the direction from right to left we assume unsatisfiability of φ and show termination of $\mathcal{U}^\varphi$. For this purpose we apply semantic labeling. Note that we consider $\mathcal{U}^\varphi$ as one-sorted. The idea is to label the symbol unsat by the value which φ evaluates to—under all possible assignments. To obtain a model, the function symbols are interpreted in the Boolean algebra. The interpretation $\mathcal{B}$ over the carrier $\mathbb{B}$ satisfying $\bot_\mathcal{B} = 0$, $\top_\mathcal{B} = 1$, $\star_\mathcal{B}(x, y) = x \cdot y$, $-_\mathcal{B}(x) = \overline{x}$, and $\mathsf{unsat}_\mathcal{B}(p_1, \ldots, p_n, y) = 0$ is a model for $\mathcal{U}^\varphi$. Next the labeling for $\mathcal{U}^\varphi$ is defined. For this purpose only the function symbol unsat gets labeled, i.e., $L_\star = L_- = \varnothing$[1] and $L_\mathsf{unsat} = \mathbb{B}$. The labeling function $\ell_\mathsf{unsat} \colon \mathbb{B}^{n+1} \to \mathbb{B}$ is defined as: $\ell_\mathsf{unsat}(p_1, \ldots, p_n, y) = y$. By assumption the formula φ evaluates to 0 under all assignments. Hence the labeled variant of rule 7.1 looks like

$$\mathsf{unsat}_1(p_1, \ldots, p_n, \top) \to \mathsf{unsat}_0(p_1, \ldots, p_n, \ulcorner\varphi\urcorner). \tag{7.2}$$

Termination of the labeled system can then easily be shown by some basic method, e.g., LPO; choosing the precedence $\mathsf{unsat}_1 > \mathsf{unsat}_0, \star, -$ allows to orient rule 7.2 from left to right and $- > \bot, \top$ handles the rules in $\mathcal{S}\mathsf{imp}$. So $\mathcal{U}^\varphi_\mathsf{lab}$ is terminating. Theorem 7.1 yields the termination of $\mathcal{U}^\varphi$. □

[1] Labeling constants is superfluous and hence we implicitly set $L_\bot = L_\top = \varnothing$.

7.3 Transforming Satisfiability to Termination

In the previous section the task was somehow simpler since there it sufficed to construct a non-terminating sequence if there exists a satisfying assignment. Hence by guessing a satisfying assignment for φ one could construct an infinite sequence in the TRS $\mathcal{U}^\varphi$. In this section the endeavor is more challenging, because one has to guarantee that one cycles if *no* satisfying assignment exists. Hence, the parametrized TRS will have to test all assignments before entering a loop if none of them satisfied the formula φ. Thus we have to provide the possibility to generate all assignments successively. The following three rules, referred to as $\mathcal{N}\mathsf{ext}$ do this job by representing assignments as bit-lists (consequently the signature $\mathcal{F}$ is extended by the binary function symbol $(::) : \mathsf{bool} \times \mathsf{list} \to \mathsf{list}$, the constant $\mathsf{nil} : \mathsf{list}$, and the unary function symbol $\mathsf{next} : \mathsf{list} \to \mathsf{list}$):

$$\begin{aligned} \mathsf{next}(\mathsf{nil}) &\to \mathsf{nil} \\ \mathsf{next}(\bot :: xs) &\to \top :: xs \\ \mathsf{next}(\top :: xs) &\to \bot :: \mathsf{next}(xs) \end{aligned}$$

To ease notation we will encode lists over $\bot$ and $\top$ as natural numbers. Therefore, lists are interpreted as little endian representation of binary numbers where $\bot$ corresponds to 0 and $\top$ to 1. Let $\mathcal{G}$ be the signature $\{\bot, \top, ::, \mathsf{nil}\}$. The mapping $\mathsf{enc}\colon \mathcal{T}(\mathcal{G}) \to \mathbb{N} \times \mathbb{N}$, $\mathsf{enc}(\mathsf{nil}) = (0,0)$ and $\mathsf{enc}(x :: xs) = (x + 2i, l + 1)$ where $\mathsf{enc}(xs) = (i,l)$, uniquely associates lists with entries $\bot$ or $\top$ to pairs. The first component of the pair is the little endian representation of the bit-list whereas the second component is the length of the list. For convenience we denote (i,l) by $\mathbf{i_l}$. Furthermore if l is irrelevant or fixed we feel free to omit it. Taking the above conventions into account the bit-list $[\top; \bot; \top; \top]$[2] can be written as $\mathbf{13_4}$ or more sloppily as $\mathbf{13}$. But we do not only identify these bit-lists with natural numbers, they also encode substitutions. Hence, a bit-list $[t_1; \ldots; t_n]$ gives rise to a substitution σ with $\sigma(p_i) = t_i$ for $1 \leqslant i \leqslant n$. Using this convention a term t indexed with a bold face integer denotes the result of applying the substitution to the term, i.e., $(p_1 \star ((- p_2) \star p_3))_{\mathbf{13}}$ denotes $\top \star ((- \bot) \star \top)$.

Lemma 7.5. *For a bit-list t, $\mathsf{next}(t)$ rewrites to the successor of t:*

$$\textit{If } \mathsf{enc}(t) = \mathbf{i_l} \textit{ then } \mathsf{next}(t) \to^*_{\mathcal{N}\mathsf{ext}} t' \textit{ with } \mathsf{enc}(t') = (\mathbf{i} + \mathbf{1} \bmod \mathbf{2^l})_\mathbf{l}.$$

Proof. By induction on the structure of t and unfolding the definition of $\mathbf{i_l}$. □

To proceed we explicitly state the function signature sig, i.e., the sort for each function symbol, in the left column of Table 7.1. In the theorem below the variables $p_1, \ldots, p_n$ are of sort bool and xs is of sort list.

Theorem 7.6. *Let $\varphi \in \mathcal{P}(\mathcal{A}_n)$. Then the generic $\{\mathsf{bool}, \mathsf{list}\}$-sorted TRS $\mathcal{S}^\varphi$ that contains all rules of $\mathcal{S}\mathsf{imp}$, $\mathcal{N}\mathsf{ext}$, and additionally*

$$\mathsf{sat}([p_1; \ldots; p_n], \bot) \to \mathsf{sat}(\mathsf{next}([p_1; \ldots; p_n]), \ulcorner \varphi \urcorner) \tag{7.3}$$

is terminating if and only if the formula φ is satisfiable.

[2] To ease readability, lists $x :: (y :: (z :: \mathsf{nil}))$ are abbreviated by $[x; y; z]$.

Table 7.1: A model for the $\{\mathsf{bool}, \mathsf{list}\}$-sorted TRS $\mathcal{S}^{\varphi}$

$\bot : \mathsf{bool}$	$\bot_{\mathcal{A}} = 0$
$\top : \mathsf{bool}$	$\top_{\mathcal{A}} = 1$
$\star : \mathsf{bool} \times \mathsf{bool} \to \mathsf{bool}$	$\star_{\mathcal{A}}(x, y) = x \cdot y$
$- : \mathsf{bool} \to \mathsf{bool}$	$-_{\mathcal{A}}(x) = \overline{x}$
$\mathsf{nil} : \mathsf{list}$	$\mathsf{nil}_{\mathcal{A}} = (0, 0)$
$:: : \mathsf{bool} \times \mathsf{list} \to \mathsf{list}$	$::_{\mathcal{A}}(x, (i, l)) = (x + 2i, l + 1)$
$\mathsf{next} : \mathsf{list} \to \mathsf{list}$	$\mathsf{next}_{\mathcal{A}}((i, l)) = (i + 1 \bmod 2^l, l)$
$\mathsf{sat} : \mathsf{list} \times \mathsf{bool} \to \mathsf{bool}$	$\mathsf{sat}_{\mathcal{A}}((i, l), b) = 0$

Proof. For the direction from left to right assume for the sake of a contradiction unsatisfiability of φ. The cyclic reduction

$$\begin{aligned}\mathsf{sat}(\mathbf{0}, \bot) \to\ & \mathsf{sat}(\mathsf{next}(\mathbf{0}), \ulcorner\varphi\urcorner_{\mathbf{0}}) \to^* \mathsf{sat}(\mathbf{1}, \ulcorner\varphi\urcorner_{\mathbf{0}}) \to^* \mathsf{sat}(\mathbf{1}, \bot) \to^* \cdots \\ & \to^* \mathsf{sat}(\mathsf{next}(\mathbf{2^n - 1}), \ulcorner\varphi\urcorner_{\mathbf{2^n - 1}}) \to^* \mathsf{sat}(\mathbf{0}, \ulcorner\varphi\urcorner_{\mathbf{2^n - 1}}) \\ & \to^* \mathsf{sat}(\mathbf{0}, \bot)\end{aligned}$$

proves non-termination of $\mathcal{S}^{\varphi}$ where $\mathsf{next}(\mathbf{i_n}) \to^*_{\mathcal{N}\mathsf{ext}} (\mathbf{i} + \mathbf{1} \bmod \mathbf{2^n})_{\mathbf{n}}$ follows from Lemma 7.5 and since we assumed that φ is unsatisfiable $\ulcorner\varphi\urcorner_{\mathbf{i_n}} \to^*_{\mathcal{S}\mathsf{imp}} \bot$ by Lemma 7.2, for all $0 \leqslant i < 2^n$.

For the direction from right to left we will again give a proof using semantic labeling. The difference this time is that we exploit the many-sorted version of semantic labeling. Now for every sort $s \in \{\mathsf{bool}, \mathsf{list}\}$ we have to specify a carrier. The choices are $A_{\mathsf{bool}} = \mathbb{B}$ and $A_{\mathsf{list}} = P := \{(i, l) \in \mathbb{N} \times \mathbb{N} \mid i < 2^l\}$. Then the interpretation in the right column of Table 7.1 is a model for $\mathcal{S}^{\varphi}$. We show this for the $\mathcal{N}\mathsf{ext}$-rules. Let us fix an arbitrary value $(i, l) \in P$ for xs. The three $\mathcal{N}\mathsf{ext}$-rules generate the three equalities

$$(0 + 1 \bmod 2^0, 0) = (0, 0) \tag{7.4}$$

$$((0 + 2i) + 1 \bmod 2^{l+1}, l + 1) = (1 + 2i, l + 1) \tag{7.5}$$

$$((1 + 2i) + 1 \bmod 2^{l+1}, l + 1) = (0 + 2(i + 1 \bmod 2^l), l + 1) \tag{7.6}$$

Equation (7.4) is trivially valid. Since $i < 2^l$ by definition of P equation (7.5) holds since the modulo operation can be omitted. Validity of equation (7.6) is shown by case distinction. Considering the case when $i = 2^l - 1$ it simplifies to $1 + 2(2^l - 1) + 1 \bmod 2^{l+1} = 0 + 2(2^l - 1 + 1 \bmod 2^l)$ where both sides evaluate to 0. For the other case we know that $i < 2^l - 1$ and consequently $2i + 2 < 2^{l+1}$. Hence, the modulo operation does no harm and both sides evaluate to the same value.

The following sets of labels are employed: $L_{\star} = L_{-} = L_{::} = L_{\mathsf{next}} = \varnothing$ and $L_{\mathsf{sat}} = \mathbb{N} \times \mathbb{B}$. Then, the labeling function $\ell_{\mathsf{sat}} \colon P \times \mathbb{N} \to \mathbb{N} \times \mathbb{B}$ with

$\ell_{\mathsf{sat}}((i,l),b) = (i,b)$ is used which produces the following labeled variants of rule 7.3

$$\mathsf{sat}_{i,0}([p_1;\ldots;p_n],\bot) \to \mathsf{sat}_{(i+1 \bmod 2^n),\varphi_{\mathbf{i_n}}}(\mathsf{next}([p_1;\ldots;p_n]),\ulcorner\varphi\urcorner) \tag{7.7}$$

where $0 \leqslant i < 2^n$. In the right-hand side of the generic rule 7.7 the expression $\varphi_{\mathbf{i_n}}$ means that φ is evaluated by the assignment corresponding to the bit-list $\mathbf{i_n}$.

If for at least one assignment φ evaluates to 1 then the system can be proved terminating. Assume that the j-th assignment satisfies φ. Then the precedence

$$\begin{aligned}
&\mathsf{sat}_{(j+1),0} > \mathsf{sat}_{(j+2),0} > \ldots > \mathsf{sat}_{(2^n-1),0} > \mathsf{sat}_{0,0} > \mathsf{sat}_{1,0} > \ldots > \mathsf{sat}_{j,0}\\
&\quad \mathsf{sat}_{i,0} > \mathsf{sat}_{(i+1 \bmod 2^n),1} \quad (0 \leqslant i < 2^n)\\
&\quad \mathsf{sat}_{j,0} > \mathsf{next}, \star, - > \bot, \top
\end{aligned}$$

is well-founded and allows LPO to orient all rules of the labeled TRS $\mathcal{S}^{\varphi}_{\mathsf{lab}}$ from left to right. Termination of $\mathcal{S}^{\varphi}$ follows from Theorem 7.1. □

As an example consider the transformation of the formula $p_1 \wedge \neg p_2$ below.

Example 7.7. The system $\mathcal{S}^{p_1 \wedge \neg p_2}$ gives rise to the labeled rules

$$\begin{aligned}
\mathsf{sat}_{0,0}([p_1;p_2],\bot) &\to \mathsf{sat}_{1,0}(\mathsf{next}([p_1;p_2]), p_1 \star (-p_2))\\
\mathsf{sat}_{1,0}([p_1;p_2],\bot) &\to \mathsf{sat}_{2,1}(\mathsf{next}([p_1;p_2]), p_1 \star (-p_2))\\
\mathsf{sat}_{2,0}([p_1;p_2],\bot) &\to \mathsf{sat}_{3,0}(\mathsf{next}([p_1;p_2]), p_1 \star (-p_2))\\
\mathsf{sat}_{3,0}([p_1;p_2],\bot) &\to \mathsf{sat}_{0,0}(\mathsf{next}([p_1;p_2]), p_1 \star (-p_2)).
\end{aligned}$$

Note that because in the second line the term $(p_1 \star (-p_2))_{\mathbf{1}}$ is interpreted as 1, the system can easily be proved terminating by LPO with the precedence

$$\mathsf{sat}_{2,0} > \mathsf{sat}_{3,0} > \mathsf{sat}_{0,0} > \mathsf{sat}_{1,0} > \mathsf{sat}_{2,1}, \mathsf{next}, \star, - \qquad - > \bot, \top.$$

In this translation the TRS $S^{\varphi}_{\mathsf{lab}}$ gets exponentially larger (in the number of variables in φ) than the original unlabeled system. More precisely, rule 7.3 gives rise to 2^n different labeled variants due to the n Boolean variables in the list $[p_1;\ldots;p_n]$. But the resulting TRS is still finite, in contrast to the one from the next subsection.

7.3.1 An Alternative Transformation

In the transformation $\mathcal{S}^{\varphi}$ the formula φ gets assigned the values implicitly by pattern matching because the same variables $p_1,\ldots,p_n$ are used in the formula and in the assignment. One not so nice side-effect is that in rule 7.3 the list of variables occurring in φ must be specified as the first argument to sat. Here we present a different translation where the variables $p_1,\ldots,p_n$ are considered as constants $\mathsf{v_1},\ldots,\mathsf{v_n}$ in the signature $\mathcal{F}$. For terms that represent formulas on the syntactic level, the sort $\mathsf{formula}$ is used, i.e., $\mathsf{v_i} : \mathsf{formula}$ for $1 \leqslant i \leqslant n$. Furthermore a close inspection of the systems $\mathcal{U}^{\varphi}$ and $\mathcal{S}^{\varphi}$ reveals that there is no clear separation between syntax and semantics when formulas are represented

as terms. To differentiate these two concepts we employ different function symbols for the two layers. Once more the signature $\mathcal{F}$ is augmented by a binary function symbol $\mathsf{and} : \mathsf{formula} \times \mathsf{formula} \to \mathsf{formula}$ and a unary function symbol $\mathsf{not} : \mathsf{formula} \to \mathsf{formula}$. Consequently also the encoding $\ulcorner * \urcorner$ must now map formulas to their syntactic representation on the term level. Hence the function $\ulcorner * \urcorner$ is redefined accordingly, i.e., $\ulcorner * \urcorner \colon \mathcal{P}(\mathcal{A}_n) \to \mathcal{T}(\{\mathsf{v}_1, \ldots, \mathsf{v}_n, \mathsf{and}, \mathsf{not}\})$ with $\ulcorner p_i \urcorner = \mathsf{v}_i$ for $1 \leqslant i \leqslant n$, $\ulcorner \varphi \wedge \psi \urcorner = \mathsf{and}(\ulcorner \varphi \urcorner, \ulcorner \psi \urcorner)$, and $\ulcorner \neg \varphi \urcorner = \mathsf{not}(\ulcorner \varphi \urcorner)$. Thus, for the formula $p_1 \wedge \neg p_2$ the (syntactic) term representation $\ulcorner \varphi \urcorner$ is $\mathsf{and}(\mathsf{v}_1, \mathsf{not}(\mathsf{v}_2))$.

In the TRS $\mathcal{S}^{\varphi}$ the assignment was applied automatically by pattern matching of the variables. Now we employ separate rewrite rules that perform that step. Note that these rules at the same time execute the transformation from the syntactic to the semantic level. The TRS $\mathcal{A}\mathsf{ssign}$

$$\begin{aligned}
\mathsf{assign}(xs, \mathsf{and}(x, y)) &\to \mathsf{assign}(xs, x) \star \mathsf{assign}(xs, y) \\
\mathsf{assign}(xs, \mathsf{not}(x)) &\to - \mathsf{assign}(xs, x) \\
\mathsf{assign}(xs, \mathsf{v_i}) &\to \mathsf{nth}(xs, \mathsf{s}^i(\mathsf{0})) && 1 \leqslant i \leqslant n \\
\mathsf{nth}(\bot :: xs, \mathsf{0}) &\to \bot \\
\mathsf{nth}(\top :: xs, \mathsf{0}) &\to \top \\
\mathsf{nth}(b :: xs, \mathsf{s}(j)) &\to \mathsf{nth}(xs, j)
\end{aligned}$$

performs the task of $[\alpha]$ on the term representation of propositional formulas. The way how assignments were generated in the previous subsection is no longer suitable. There all variables occurring in φ had to be specified in sat's first argument. Since we want to get rid of that requirement the idea is to start with an empty assignment (empty list) and increase its length repeatedly. Hence in this section the assignments are no longer computed modulo some length but the *overflow* is simply taken into account by increasing the length of the list. The three rules below are referred to as the TRS $\mathcal{N}\mathsf{ext2}$:

$$\begin{aligned}
\mathsf{next}(\mathsf{nil}) &\to \top :: \mathsf{nil} \\
\mathsf{next}(\bot :: xs) &\to \top :: xs \\
\mathsf{next}(\top :: xs) &\to \bot :: \mathsf{next}(xs)
\end{aligned}$$

Similar to before a more readable notation for bit-lists is employed, i.e., they are identified with natural numbers as follows: $\mathsf{enc} \colon \mathcal{T}(\mathcal{G}) \to \mathbb{N}$ with $\mathsf{enc}(\mathsf{nil}) = \mathsf{enc}(\bot) = 0$, $\mathsf{enc}(\top) = 1$, and $\mathsf{enc}(x :: xs) = \mathsf{enc}(x) + 2\,\mathsf{enc}(xs)$. This encoding is not injective because the lists $[\top; \bot; \bot]$ and $[\top]$ are both denoted by $\mathbf{1}$. In our setting these (more or less) leading zeros do not pose a problem.

Lemma 7.8. *For a bit-list t, $\mathsf{next}(t)$ rewrites to the successor of t:*

$$\textit{If } \mathsf{enc}(t) = \mathbf{i} \textit{ then } \mathsf{next}(t) \to^*_{\mathcal{N}\mathsf{ext2}} t' \textit{ with } \mathsf{enc}(t') = \mathbf{i} + \mathbf{1}.$$

Proof. By induction on the structure of t and unfolding the definition of $\mathbf{i}$. □

The desired property that the rules in $\mathcal{A}\mathsf{ssign}$ evaluate the term representation $\ulcorner \varphi \urcorner$ for a given bit-list $\mathbf{i}$ is formalized in the lemma below.

Table 7.2: A model for the $\{\mathsf{bool}, \mathsf{formula}, \mathsf{list}, \mathsf{nat}\}$-sorted TRS $\mathcal{T}^{\varphi}$

$\bot : \mathsf{bool}$	$\top_{\mathcal{A}} = 1$
$\top : \mathsf{bool}$	$\bot_{\mathcal{A}} = 0$
$\star : \mathsf{bool} \times \mathsf{bool} \to \mathsf{bool}$	$\star_{\mathcal{A}}(x, y) = x \cdot y$
$- : \mathsf{bool} \to \mathsf{bool}$	$-_{\mathcal{A}}(x) = \overline{x}$
$\mathsf{nil} : \mathsf{list}$	$\mathsf{nil}_{\mathcal{A}} = 0$
$:: : \mathsf{bool} \times \mathsf{list} \to \mathsf{list}$	$::_{\mathcal{A}}(x, i)) = x + 2i$
$\mathsf{next} : \mathsf{list} \to \mathsf{list}$	$\mathsf{next}_{\mathcal{A}}(i) = i + 1$
$\mathsf{v_i} : \mathsf{formula} \quad 1 \leqslant i \leqslant n$	$\mathsf{v_i}_{\mathcal{A}} = p_i$
$\mathsf{and} : \mathsf{formula} \times \mathsf{formula} \to \mathsf{formula}$	$\mathsf{and}_{\mathcal{A}}(x, y) = x \wedge y$
$\mathsf{not} : \mathsf{formula} \to \mathsf{formula}$	$\mathsf{not}_{\mathcal{A}}(x) = \neg x$
$0 : \mathsf{nat}$	$0_{\mathcal{A}} = 0$
$\mathsf{s} : \mathsf{nat} \to \mathsf{nat}$	$\mathsf{s}_{\mathcal{A}}(x) = x + 1$
$\mathsf{assign} : \mathsf{list} \times \mathsf{formula} \to \mathsf{bool}$	$\mathsf{assign}_{\mathcal{A}}(i, \varphi) = [\alpha_i](\varphi)$
$\mathsf{nth} : \mathsf{list} \times \mathsf{nat} \to \mathsf{bool}$	$\mathsf{nth}_{\mathcal{A}}(i, j) = \alpha_i(p_j)$
$\mathsf{sat} : \mathsf{list} \times \mathsf{bool} \to \mathsf{bool}$	$\mathsf{sat}_{\mathcal{A}}(i, b) = 0$

Lemma 7.9. *Let $\varphi \in \mathcal{P}(\mathcal{A}_n)$ and let $\mathbf{i}$ be the encoding of an assignment α with $[\alpha](\varphi) = 0$. Then $\mathsf{assign}(\mathbf{i}, \ulcorner\varphi\urcorner) \to^*_{\mathcal{A}\mathsf{ssign} \cup \mathcal{S}\mathsf{imp}} \bot$.*

Proof. By induction on the structure of $\ulcorner\varphi\urcorner$ and unfolding the definition of $\mathbf{i}$. □

Now, we will establish a theorem similar to Theorem 7.6. Again, we prove the theorem in a many-sorted setting. The full information is depicted in Table 7.2. The variables in the TRS are associated to sorts as follows: $b \in \mathcal{V}_{\mathsf{bool}}$, $xs \in \mathcal{V}_{\mathsf{list}}$, $j \in \mathcal{V}_{\mathsf{nat}}$, and $x, y \in \mathcal{V}_{\mathsf{formula}}$.

Theorem 7.10. *Let $\varphi \in \mathcal{P}(\mathcal{A}_n)$. Then the generic $\{\mathsf{bool}, \mathsf{formula}, \mathsf{list}, \mathsf{nat}\}$-sorted TRS $\mathcal{T}^{\varphi}$ consisting of the $\mathcal{S}$imp-, $\mathcal{N}$ext2-, and $\mathcal{A}$ssign-rules plus additionally*

$$\mathsf{sat}(xs, \bot) \to \mathsf{sat}(\mathsf{next}(xs), \mathsf{assign}(xs, \ulcorner\varphi\urcorner)) \tag{7.8}$$

is terminating if and only if φ is satisfiable.

Proof. Concerning the direction from left to right one can again construct a non-terminating reduction for any unsatisfiable formula φ. In order not to get stuck while evaluating $\mathsf{assign}(\mathbf{i}, \ulcorner\varphi\urcorner)$ a sufficiently large $\mathbf{i}$ is taken (e.g., $\mathbf{i} = 2^{n+1}$). Then there is the infinite sequence

$$\begin{aligned}\mathsf{sat}(\mathbf{i}, \bot) \to\ & \mathsf{sat}(\mathsf{next}(\mathbf{i}), \mathsf{assign}(\mathbf{i}, \ulcorner\varphi\urcorner)) \\ \to^*\ & \mathsf{sat}(\mathbf{i}+\mathbf{1}, \bot) \to \mathsf{sat}(\mathsf{next}(\mathbf{i}+\mathbf{1}), \mathsf{assign}(\mathbf{i}+\mathbf{1}, \ulcorner\varphi\urcorner)) \\ \to^*\ & \mathsf{sat}(\mathbf{i}+\mathbf{2}, \bot) \to \mathsf{sat}(\mathsf{next}(\mathbf{i}+\mathbf{2}), \mathsf{assign}(\mathbf{i}+\mathbf{2}, \ulcorner\varphi\urcorner)) \to^* \cdots\end{aligned}$$

where the $\to$-steps are applications of rule 7.8 and the $\to^*$-steps can be performed because of Lemmata 7.8 and 7.9.

For the direction from right to left again a semantic labeling approach is followed. The interpretation from Table 7.2 models the $\{\mathsf{bool}, \mathsf{formula}, \mathsf{list}, \mathsf{nat}\}$-sorted TRS $\mathcal{T}^\varphi$. What remains to be defined is an enumeration α_i of assignments as follows: $\alpha_i(p_j) = f^j(i) \bmod 2$ with $f^0(i) = i$ and $f^{j+1}(i) = f^j(\lceil i \div 2 \rceil)$. Checking that $\mathcal{A}$ models $\mathcal{T}^\varphi$ is straightforward.

Again, only the function symbol sat is labeled. Note that the labeled TRS T^φ_{lab} is infinite since all possible instances of bit-lists are considered (compared to finitely many bit-lists of a specified length in the previous subsection). The labeling function $\ell_{\mathsf{sat}}(i, b) = (i, b)$ gives rise to infinitely many rules of the following structure

$$\mathsf{sat}_{i,0}(xs, \bot) \to \mathsf{sat}_{(i+1),[\alpha_i](\varphi)}(\mathsf{next}(xs), \mathsf{assign}(xs, \ulcorner\varphi\urcorner)).$$

Similar to before a precedence of the shape $\mathsf{sat}_{i,0} > \mathsf{sat}_{(i+1),0}$ if $[\alpha_i](\varphi) = 0$ and $\mathsf{sat}_{i,0} > \mathsf{sat}_{(i+1),1}$ if $[\alpha_i](\varphi) = 1$ for all $i \geqslant 0$ is needed which in general might not be well-founded since it can contain the infinite sequence

$$\mathsf{sat}_{0,0} > \mathsf{sat}_{1,0} > \mathsf{sat}_{2,0} > \cdots$$

but due to the assumption that φ is satisfiable, not all of these precedence comparisons are necessary. If $[\alpha_j](\varphi) = 1$ then there is no labeled rule which demands $\mathsf{sat}_{j,0} > \mathsf{sat}_{(j+1),0}$. Without any loss of generality we can assume $0 \leqslant j < 2^n$. Due to the construction of α_j also α_{j+2^n}, $\alpha_{j+2^{n+1}}$, $\ldots$ satisfy φ and hence removing all superfluous comparisons $\mathsf{sat}_{(j+2^{n+m}),0} > \mathsf{sat}_{(j+2^{n+m}+1),0}$ for all $m \in \mathbb{N}$ produces a well-founded precedence (because for any $i \in \mathbb{N}$ one can find a $k \in \mathbb{N}$ such that $i \leqslant j + 2^{n+k}$). It follows that $\mathcal{T}^\varphi_{\mathsf{lab}}$ is terminating. Termination of $\mathcal{T}^\varphi$ follows from Theorem 7.1. □

Although the transformations $\mathcal{S}^\varphi$ and $\mathcal{T}^\varphi$ look very similar at first, they are quite different. Concerning the number of rewrite rules, $\mathcal{S}^\varphi$ does not depend on φ whereas $\mathcal{T}^\varphi$ depends linearly on the number of variables in φ. On the other hand, the list of variables $p_1, \ldots, p_n$ must be given as an argument to sat in $\mathcal{S}^\varphi$. In Section 7.4 it becomes apparent that proving (non-)termination automatically is much more challenging for $\mathcal{T}^\varphi$ than for $\mathcal{S}^\varphi$. The main reason is that by separating syntax from semantics, there is less structure that can be exploited by termination tools. The non-termination proofs become more challenging because for $\mathcal{S}^\varphi$ an infinite rewrite sequence can be captured by considering cyclic reductions of ground terms, i.e., $t \to^+ t$ for a ground term t (cf. the proof of Theorem 7.6). In contrast $\mathcal{T}^\varphi$ really demands looping reductions, i.e., $t \to^+ C[t\sigma]$ where the context C is empty but t may no longer be ground since the lengths of the bit-lists are increased.

7.4 Experiments

For experimental results we considered all automated (non-)termination analyzers that participated in the 2007 or 2008 editions of the international termination competition for term rewrite systems augmented with TPA [49], a

Table 7.3: Experimental Results

	2 vars, depth 3			3 vars, depth 4			4 vars, depth 5		
	$\mathcal{S}^\varphi$	$\mathcal{T}^\varphi$	$\mathcal{U}^\varphi$	$\mathcal{S}^\varphi$	$\mathcal{T}^\varphi$	$\mathcal{U}^\varphi$	$\mathcal{S}^\varphi$	$\mathcal{T}^\varphi$	$\mathcal{U}^\varphi$
	T/ N	T/N	T/ N	T/N	T/N	T/ N	T/N	T/N	T/ N
AProVE	81/19	0/ 0	19/81	34/ 0	0/ 0	10/88	14/ 0	0/ 0	5/79
Jambox	16/ 0	0/ 0	19/ 0	24/ 0	0/ 0	12/ 0	15/ 0	0/ 0	11/ 0
NTI	0/19	0/ 0	0/81	0/ 5	0/ 0	0/74	0/ 0	0/ 0	0/11
TPA	0/ 0	0/ 0	1/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0	0/ 0
T$_\mathsf{T}$T$_2$	10/ 0	0/ 0	0/ 0	6/ 0	0/ 0	0/ 0	5/ 0	0/ 0	0/ 0

tool with strong support for termination proofs via semantic labeling. To our knowledge none of these tools supports analysis of sorted TRSs. Consequently we provide our examples unsorted. As already stated in the beginning, dropping sorts does not affect termination of the TRSs we propose. Furthermore we stress that the proofs of Theorems 7.6 and 7.10 can be modified to work on unsorted TRSs. For the TRS $\mathcal{S}^\varphi$ this means that the interpretations range over the set of pairs P whereas the proof of Theorem 7.10 can be generalized to one sort by using the natural numbers as a carrier and representing formulas via a Gödel encoding [35].

It turned out that even for rather small formulas (some of) our transformations produce rewrite systems whose termination analysis is challenging. We considered 100 randomly generated formulas of different shapes. Table 7.3 summarizes the results, e.g., formulas of depth three using two different propositional variables are considered in the leftmost block, etc. Every tool was run on all TRSs resulting from transforming the formula φ to $\mathcal{S}^\varphi$, $\mathcal{T}^\varphi$, and $\mathcal{U}^\varphi$ for at most 60 seconds to analyze termination (T) or non-termination (N) of each system. Globally speaking, for TRSs originating from very small formulas AProVE [30] performs best. This is due to its support for narrowing which allows to exploit the structure of $\mathcal{S}^\varphi$ and $\mathcal{U}^\varphi$. Jambox [21] solves some instances by semantic labeling over Boolean models (which is very close to the way how we proved termination) and by the matrix method. The latter systems could also be handled by T$_\mathsf{T}$T$_2$ [56]. NTI [71] supports only non-termination analysis, using an unfolding operator. Semantic labeling based on Boolean models and (quasi-)models over the naturals is implemented in TPA [52, 50] which usually performs very well on standard examples. The experiments reveal that the latter is not powerful for the systems obtained from the transformations proposed in this chapter.

But narrowing is expensive which can be seen by comparing the different blocks of Table 7.3. AProVE can handle all TRSs resulting from the $\mathcal{S}^\varphi$ translation if formulas are of depth three but for depth four (five) the performance decreases to 34% (14%). For the other translations the effect is not so tremendous, well, for $\mathcal{T}^\varphi$ the surprising outcome is that no tool could handle any system at all and the systems in $\mathcal{U}^\varphi$ are generally a bit easier since they do not iterate over the assignments. Needless to say, the formulas φ which are

considered for our experiments are a very trivial task for any SAT solver.

We conclude this section by a sketch of how AProVE solves many instances by considering the TRS S^φ for $\varphi = p_1 \wedge p_2$. After some preliminary analysis based on dependency pairs, AProVE concludes that any infinite sequence applies the rule

$$\mathsf{sat}([p_1; p_2], \bot) \to \mathsf{sat}(\mathsf{next}([p_1; p_2]), p_1 \star p_2)$$

indefinitely. Narrowing the above rule at position 1 allows to replace it by the two rules

$$\mathsf{sat}([\bot; p_2], \bot) \to \mathsf{sat}(\mathsf{next}([\bot; p_2]), \bot \star p_2)$$
$$\mathsf{sat}([\top; p_2], \bot) \to \mathsf{sat}(\mathsf{next}([\top; p_2]), \top \star p_2)$$

and narrowing these rules at position 1 gives

$$\mathsf{sat}([\bot; \bot], \bot) \to \mathsf{sat}(\mathsf{next}([\bot; \bot]), \bot \star \bot)$$
$$\mathsf{sat}([\bot; \top], \bot) \to \mathsf{sat}(\mathsf{next}([\bot; \top]), \bot \star \top)$$
$$\mathsf{sat}([\top; \bot], \bot) \to \mathsf{sat}(\mathsf{next}([\top; \bot]), \top \star \bot)$$
$$\mathsf{sat}([\top; \top], \bot) \to \mathsf{sat}(\mathsf{next}([\top; \top]), \top \star \top).$$

After this state is reached the right-hand sides can be rewritten [31] using the $\mathcal{S}\mathsf{imp}$ and $\mathcal{N}\mathsf{ext}$ rules which allows the dependency graph processor to conclude termination.

7.5 Summary

In this chapter we proposed three different transformations from propositional formulas φ to confluent—since orthogonal—term rewrite systems $\mathcal{S}^\varphi$, $\mathcal{T}^\varphi$, and $\mathcal{U}^\varphi$ such that φ is satisfiable (unsatisfiable) if and only if $\mathcal{S}^\varphi$, $\mathcal{T}^\varphi$ ($\mathcal{U}^\varphi$) is terminating. Although the systems can be proved (non-)terminating by semantic labeling using intuitive models, state-of-the-art termination tools fail even on very small and simple TRSs. Especially the transformation $\mathcal{T}^\varphi$ produces unsolvable rewrite systems which might be due to the fact that it preserves much less structure than $\mathcal{S}^\varphi$ does. If tool authors investigate the reasons why the generated problems are that hard, new termination techniques could emerge.

Conclusion

This thesis showed that SAT solving—or more general, constraint solving, since also PB and SMT are considered—can be used very successfully for termination analysis in rewriting. One benefit of employing constraint solvers is that traditional methods can not only be implemented with much less effort but additionally these encodings outperform dedicated algorithms in performance.

This tremendous speedup was shown in Chapter 2 by means of KBO where additionally the flexibility of the encodings is stressed, i.e., it is possible to produce easily human readable proofs by, e.g., minimizing weights. Additionally we introduced a method to compute upper bounds for the weights which yields an alternative decision procedure for KBO.

Another major advantage of employing constraint solvers is that due to the expressiveness of their input language completely new termination criteria can be invented and implemented. Increasing interpretations have been demonstrated to be such a case where in parallel suitable polynomial interpretations are searched while performing a cycle analysis in the dependency graph. Needless to say, increasing interpretations subsume the setting of traditional polynomial interpretations.

The chapter devoted to matrix interpretations generalized the underlying theory to allow also non-natural coefficients in the matrices. Constraint solving was also employed there to find these—possibly real-valued—coefficients. To the author's knowledge this is the first work that considers (a fragment of) real numbers in SAT. Experimental results demonstrate the applicability of the approach.

That not only termination analysis is in the reach of constraint solvers was demonstrated in Chapter 5. Here, a looping reduction within an SRS is represented as a set of constraints and from a satisfying assignment a loop can easily be reconstructed. Additionally in this chapter we formalized non-termination in the theorem prover Isabelle and integrated the corresponding check-functions into CeTA which is now capable of certifying loops.

Chapter 6 addressed how to solve arithmetic constraints efficiently by means of SAT and SMT solvers.

And finally—after all these (non-)termination encodings in SAT, PB, or SMT—we investigated the other direction, i.e., encoding propositional satisfiability as a termination problem in rewriting. The resulting transformations yield termination problems which are out of the reach of state-of-the-art termination tools, hence they are suitable of providing large testbeds of challenging problems.

Bibliography

[1] Alarcón, B., Lucas, S., Navarro-Marset, R.: Proving termination with matrix interpretations over the reals. In: Proc. of the 10th International Workshop on Termination (WST 2009), pp. 12–15 (2009)

[2] Aoto, T., Yamada, T.: Termination of simply typed term rewriting by translation and labelling. In: Proc. of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003), volume 2706 of Lecture Notes in Computer Science, pp. 380–394 (2003)

[3] Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science 236(1-2), 133–178 (2000)

[4] Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)

[5] Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions. TCS Texts in Theoretical Computer Science. An EATCS Series. Springer, Berlin-Heidelberg (2004)

[6] Blanqui, F., Delobel, W., Coupet-Grimal, S., Hinderer, S., Koprowski, A.: CoLoR, a Coq library on rewriting and termination. In: Proc. of the 8th International Workshop on Termination (WST 2006), pp. 69–73 (2006)

[7] Borralleras, C., Ferreira, M., Rubio, A.: Complete monotonic semantic path orderings. In: Proc. of the 17th International Conference on Automated Deduction (CADE 2000), volume 1831 of Lecture Notes in Artificial Intelligence, pp. 346–364 (2000)

[8] Borralleras, C., Lucas, S., Navarro-Marset, R., Rodriguez-Carbonell, E., Rubio, A.: Solving non-linear polynomial arithmetic via SAT modulo linear arithmetic. In: Proc. of the 22nd International Conference on Automated Deduction (CADE 2009), volume 5663 of Lecture Notes in Artificial Intelligence, pp. 294–305 (2009)

[9] Bryant, R.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys 24(3), 293–318 (1992)

[10] Codish, M., Lagoon, V., Stuckey, P.: Solving partial order constraints for LPO termination. In: Proc. of the 17th International Conference on Rewriting Techniques and Applications (RTA 2006), volume 4098 of Lecture Notes in Computer Science, pp. 4–18 (2006)

[11] Codish, M., Schneider-Kamp, P., Lagoon, V., Thiemann, R., Giesl, J.: SAT solving for argument filterings. In: Proc. of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2006), volume 4246 of Lecture Notes in Artificial Intelligence, pp. 30–44 (2006)

[12] Contejean, E., Courtieu, P., Forest, J., Pons, O., Urbain, X.: Certification of automated termination proofs. In: Proc. of the 6th International Symposium on Frontiers of Combining Systems (FroCoS 2007), volume 4720 of Lecture Notes in Artificial Intelligence, pp. 148–162 (2007)

[13] Cook, S.: The complexity of theorem-proving procedures. In: Proc. of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971), pp. 151–158 (1971)

[14] Courtieu, P., Forest, J., Urbain, X.: Certifying a termination criterion based on graphs, without graphs. In: Proc. of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008), volume 5170 of Lecture Notes in Computer Science, pp. 183–198 (2008)

[15] Danzig, G.: Linear Programming and Extensions. Princeton University Press, Princeton (1963)

[16] Dershowitz, N.: Termination of rewriting. Journal of Symbolic Computation 3(1-2), 69–116 (1987)

[17] Dick, J., Kalmus, J., Martin, U.: Automating the Knuth-Bendix ordering. Acta Informatica 28, 95–119 (1990)

[18] Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Proc. of the 18th International Conference on Computer Aided Verification (CAV 2006), volume 4144 of Lecture Notes in Computer Science, pp. 81–94 (2006)

[19] Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation 2(1–4), 1–26 (2006)

[20] Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), volume 2919 of Lecture Notes in Computer Science, pp. 502–518 (2004)

[21] Endrullis, J.: (Jambox). Available from `http://joerg.endrullis.de`.

[22] Endrullis, J., Waldmann, J., Zantema, H.: Matrix interpretations for proving termination of term rewriting. Journal of Automated Reasoning 40(2-3), 195–220 (2008)

[23] Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: SAT solving for termination analysis with polynomial interpretations. In: Proc. of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT 2007), volume 4501 of Lecture Notes in Computer Science, pp. 340–354 (2007)

[24] Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: Maximal termination. In: Proc. of the 19th International Conference on Rewriting Techniques and Applications (RTA 2008), volume 5117 of Lecture Notes in Computer Science, pp. 110–125 (2008)

[25] Fuhs, C., Navarro-Marset, R., Otto, C., Giesl, J., Lucas, S., Schneider-Kamp, P.: Search techniques for rational polynomial orders. In: Proc. of the 9th International Conference on Artificial Intelligence and Symbolic Computation (AISC 2008), volume 5144 of Lecture Notes in Artificial Intelligence, pp. 109–124 (2008)

[26] Gale, D.: The theory of linear economic models. McGraw-Hill, New York (1960)

[27] Gebhardt, A., Hofbauer, D., Waldmann, J.: Matrix evolutions. In: Proc. of the 9th International Workshop on Termination (WST 2007), pp. 4–8 (2007)

[28] Geser, A., Hofbauer, D., Waldmann, J., Zantema, H.: On tree automata that certify termination of left-linear term rewriting systems. Information and Computation 205(4), 512–534 (2007)

[29] Geser, A., Zantema, H.: Non-looping string rewriting. RAIRO – Theoretical Informatics and Applications 33(3), 279–302 (1999)

[30] Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic termination proofs in the dependency pair framework. In: Proc. of the 3rd International Joint Conference on Automated Reasoning (IJCAR 2006), volume 4130 of Lecture Notes in Artificial Intelligence, pp. 281–286 (2006)

[31] Giesl, J., Thiemann, R., Schneider-Kamp, P.: The dependency pair framework: Combining techniques for automated termination proofs. In: Proc. of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2004), volume 3452 of Lecture Notes in Artificial Intelligence, pp. 301–331 (2005)

[32] Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and disproving termination of higher-order functions. In: Proc. of the 5th International Symposium on Frontiers of Combining Systems (FroCoS 2005), volume 3717 of Lecture Notes in Artificial Intelligence, pp. 216–231 (2005)

[33] Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. Journal of Automated Reasoning 37(3), 155–203 (2006)

[34] Giesl, J., Swiderski, S., Schneider-Kamp, P., Thiemann, R.: Automated termination analysis for Haskell: From term rewriting to programming languages. In: Proc. of the 17th International Conference on Rewriting Techniques and Applications (RTA 2006), volume 4098 of Lecture Notes in Computer Science, pp. 297–312 (2006)

[35] Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. Monatshefte für Mathematik und Physik 38(1), 173–198 (1931)

[36] Haftmann, F.: Code generation from Isabelle/HOL theories (2008). Available from `http://isabelle.in.tum.de/doc/codegen.pdf`.

[37] Hirokawa, N., Middeldorp, A.: Polynomial interpretations with negative coefficients. In: Proc. of the 7th International Conference on Artificial Intelligence and Symbolic Computation (AISC 2004), volume 3249 of Lecture Notes in Artificial Intelligence, pp. 185–198 (2004)

[38] Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. Information and Computation 199(1-2), 172–199 (2005)

[39] Hirokawa, N., Middeldorp, A.: Tyrolean Termination Tool: Techniques and features. Information and Computation 205(4), 474–511 (2007)

[40] Hofbauer, D., Lautemann, C.: Termination proofs and the length of derivations (preliminary version). In: Proc. of the 3rd International Conference on Rewriting Techniques and Applications (RTA 1989), volume 355 of Lecture Notes in Computer Science, pp. 167–177 (1989)

[41] Hofbauer, D., Waldmann, J.: Termination of string rewriting with matrix interpretations. In: Proc. of the 17th International Conference on Rewriting Techniques and Applications (RTA 2006), volume 4098 of Lecture Notes in Computer Science, pp. 328–342 (2006)

[42] Hofbauer, D.: Termination proofs by context-dependent interpretations. In: Proc. of the 12th International Conference on Rewriting Techniques and Applications (RTA 2001), volume 2051 of Lecture Notes in Computer Science, pp. 108–121 (2001)

[43] Hofbauer, D., Waldmann, J.: Termination of $\mathsf{aa} \to \mathsf{bc}, \mathsf{bb} \to \mathsf{ac}, \mathsf{cc} \to \mathsf{ab}$. Information Processing Letters 98, 156–158 (2006)

[44] Hong, H., Jakuš, D.: Testing positiveness of polynomials. Journal of Automated Reasoning 21(1), 23–38 (1998)

[45] Kamin, S., Lévy, J.: Two generalizations of the recursive path ordering. Unpublished manuscript, University of Illinois (1980)

[46] Karmarkar, N.: A new polynomial-time algorithm for linear programming. Combinatorica 4, 373–395 (1984)

[47] Khachiyan, L.: A polynomial algorithm in linear programming. Doklady Akademia Nauk SSSR 244, 1093–1096 (1979)

[48] Knuth, D., Bendix, P.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra. Pergamon Press, New York, 263–297 (1970)

[49] Koprowski, A.: TPA: Termination proved automatically. In: Proc. of the 17th International Conference on Rewriting Techniques and Applications (RTA 2006), volume 4098 of Lecture Notes in Computer Science, pp. 257–266 (2006)

[50] Koprowski, A., Middeldorp, A.: Predictive labeling with dependency pairs using SAT. In: Proc. of the 21st International Conference on Automated Deduction (CADE 2007), volume 4603 of Lecture Notes in Artificial Intelligence, pp. 410–425 (2007)

[51] Koprowski, A., Waldmann, J.: Arctic termination ... below zero. In: Proc. of the 19th International Conference on Rewriting Techniques and Applications (RTA 2008), volume 5117 of Lecture Notes in Computer Science, pp. 202–216 (2008)

[52] Koprowski, A., Zantema, H.: Automation of recursive path ordering for infinite labelled rewrite systems. In: Proc. of the 3rd International Joint Conference on Automated Reasoning (IJCAR 2006), volume 4130 of Lecture Notes in Artificial Intelligence, pp. 332–346 (2006)

[53] Korovin, K., Voronkov., A.: Orienting rewrite rules with the Knuth-Bendix order. Information and Computation 183(2), 165–186 (2003)

[54] Korp, M., Middeldorp, A.: Proving termination of rewrite systems using bounds. In: Proc. of the 18th International Conference on Rewriting Techniques and Applications (RTA 2007), volume 4533 of Lecture Notes in Computer Science, pp. 273–287 (2007)

[55] Korp, M., Middeldorp, A.: Beyond dependency graphs. In: Proc. of the 22nd International Conference on Automated Deduction (CADE 2009), volume 5663 of Lecture Notes in Artificial Intelligence, pp. 339–354 (2009)

[56] Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean Termination Tool 2. In: Proc. of the 20th International Conference on Rewriting Techniques and Applications (RTA 2009), volume 5595 of Lecture Notes in Computer Science, pp. 295–304 (2009)

[57] Kurihara, M., Kondo, H.: Efficient BDD encodings for partial order constraints with application to expert systems in software verification. In: Proc. of the 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 2004), volume 3029 of Lecture Notes in Artificial Intelligence, pp. 827–837 (2004)

[58] Lankford, D.: On proving term rewrite systems are noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA (1979)

[59] Lepper, I.: Derivation lengths and order types of Knuth-Bendix orders. Theoretical Computer Science 269(1-2), 433–450 (2001)

[60] Löchner, B.: Things to know when implementing KBO. Journal of Automated Reasoning 36(4), 289–310 (2006)

[61] Lucas, S.: Mu-term: A tool for proving termination of context-sensitive rewriting. In: Proc. of the 15th International Conference on Rewriting Techniques and Applications (RTA 2004), volume 3091 of Lecture Notes in Computer Science, pp. 200–210 (2004)

[62] Lucas, S.: Polynomials over the reals in proofs of termination: From theory to practice. RAIRO – Theoretical Informatics and Applications 39(3), 547–586 (2005)

[63] Lucas, S.: On the relative power of polynomials with real, rational, and integer coefficients in proofs of termination of rewriting. Applicable Algebra in Engineering, Communication and Computing 17(1), 49–73 (2006)

[64] Lucas, S.: Practical use of polynomials over the reals in proofs of termination. In: Proc. of the 9th International Conference on Principles and Practice of Declarative Programming (PPDP 2007), pp. 39–50 (2007)

[65] Middeldorp, A.: Approximations for strategies and termination. Electronic Notes in Theoretical Computer Science 70(6), 1–20 (2002)

[66] Moser, G.: Derivational complexity of Knuth-Bendix orders revisited. In: Proc. of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2006), volume 4246 of Lecture Notes in Artificial Intelligence, pp. 75–89 (2006)

[67] Moser, G., Schnabl, A., Waldmann, J.: Complexity analysis of term rewriting based on matrix and context dependent interpretations. In: Proc. of the 28th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008), volume 2 of Leibniz International Proceedings in Informatics, pp. 304–315 (2008)

[68] Nguyen, M., De Schreye, D.: Polynomial interpretations as a basis for termination analysis of logic programs. In: Proc. of the 17th International Conference on Logic Programming (ICLP 2005), volume 3668 of Lecture Notes in Computer Science, pp. 311–325 (2005)

[69] Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic. Volume 2283 of Lecture Notes in Computer Science. Springer, Berlin-Heidelberg (2002)

[70] Oppelt, M.: Automatische Erkennung von Ableitungsmustern in nichtterminierenden Wortersetzungssystemen. Master's thesis, HTWK Leipzig (2008). In German.

[71] Payet, É.: Loop detection in term rewriting using the eliminating unfoldings. Theoretical Computer Science 403(2-3), 307–327 (2008)

[72] Plaisted, D., Greenbaum, S.: A structure-preserving clause form translation. Journal of Symbolic Computation 2(3), 293–304 (1986)

[73] Sato, H., Kurihara, M.: Implementation and performance evaluation of multi-completion procedures for term rewriting systems with recursive path orderings with status. IEICE Transactions on Information and Systems J89-D(4), 624–631 (2006). In Japanese.

[74] Schneider-Kamp, P., Giesl, J., Serebrenik, A., Thiemann, R.: Automated termination proofs for logic programs by term rewriting. ACM Transactions on Computational Logic 236(1-2), 133–178 (2009)

[75] Schneider-Kamp, P., Thiemann, R., Annov, E., Codish, M., Giesl, J.: Proving termination using recursive path orders and SAT solving. In: Proc. of the 6th International Symposium on Frontiers of Combining Systems (FroCoS 2007), volume 4720 of Lecture Notes in Artificial Intelligence, pp. 267–282 (2007)

[76] Steinbach, J.: Extensions and comparison of simplification orders. In: Proc. of the 3rd International Conference on Rewriting Techniques and Applications (RTA 1989), volume 355 of Lecture Notes in Computer Science, pp. 434–448 (1989)

[77] Sternagel, C., Thiemann, R., Winkler, S., Zankl, H.: CeTA – A tool for certified termination analysis. In: Proc. of the 10th International Workshop on Termination (WST 2009), pp. 56–59 (2009)

[78] Tarski, A.: A Decision Method for Elementary Algebra and Geometry. 2nd edition. University of California Press, Berkeley (1957)

[79] Thiemann, R.: The DP Framework for Proving Termination of Term Rewriting. PhD thesis, RWTH Aachen (2007). Available as technical report AIB-2007-17.

[80] Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: Proc. of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), volume 5674 of Lecture Notes in Computer Science, pp. 452–468 (2009)

[81] Thiemann, R., Sternagel, C.: Loops under strategies. In: Proc. of the 20th International Conference on Rewriting Techniques and Applications (RTA 2009), volume 5595 of Lecture Notes in Computer Science, pp. 17–31 (2009)

[82] Tseitin, G.: On the complexity of derivation in propositional calculus. In: Studies in Constructive Mathematics and Mathematical Logic, Part 2, 115–125 (1968)

[83] Waldmann, J.: Matchbox: A tool for match-bounded string rewriting. In: Proc. of the 15th International Conference on Rewriting Techniques and Applications (RTA 2004), volume 3091 of Lecture Notes in Computer Science, pp. 85–94 (2004)

[84] Waldmann, J.: Compressed loops (2007). Draft, available from `http://dfa.imn.htwk-leipzig.de/matchbox/methods/loop.pdf`.

[85] Weicker, K.: Evolutionäre Algorithmen. Teubner, Stuttgart (2002)

[86] Weiermann, A.: Termination proofs for term rewriting systems by lexicographic path orderings imply multiply recursive derivation lengths. Theoretical Computer Science 139(1-2), 355–362 (1995)

[87] Zankl, H.: BDD and SAT techniques for precedence based orders. Master's thesis, University of Innsbruck (2006)

[88] Zankl, H., Hirokawa, N., Middeldorp, A.: Constraints for argument filterings. In: Proc. of the 33rd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2007), volume 4362 of Lecture Notes in Computer Science, pp. 579–590 (2007)

[89] Zankl, H., Middeldorp, A.: Satisfying KBO constraints. In: Proc. of the 18th International Conference on Rewriting Techniques and Applications (RTA 2007), volume 4533 of Lecture Notes in Computer Science, pp. 389–403 (2007)

[90] Zankl, H., Middeldorp, A.: Increasing interpretations. In: Proc. of the 9th International Conference on Artificial Intelligence and Symbolic Computation (AISC 2008), volume 5144 of Lecture Notes in Artificial Intelligence, pp. 191–205 (2008)

[91] Zankl, H., Hirokawa, N., Middeldorp, A.: KBO orientability. Journal of Automated Reasoning 43(2), 173–201 (2009)

[92] Zankl, H., Middeldorp, A.: Increasing interpretations. Annals of Mathematics and Artificial Intelligence 56(1), 87–108 (2009)

[93] Zankl, H., Sternagel, C., Hofbauer, D., Middeldorp, A.: Finding and certifying loops. In: Proc. of the 36th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2010), Number 5901 in Lecture Notes in Computer Science, pp. 755–766 (2010)

[94] Zankl, H., Sternagel, C., Middeldorp, A.: Transforming SAT into termination of rewriting. Electronic Notes in Theoretical Computer Science 246, 199–214 (2009)

[95] Zantema, H.: Termination of term rewriting: Interpretation and type elimination. Journal of Symbolic Computation 17(1), 23–50 (1994)

[96] Zantema, H.: Termination of term rewriting by semantic labelling. Fundamenta Informaticae 24(1-2), 89–105 (1995)

[97] Zantema, H.: Reducing right-hand sides for termination. In: Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of his 60th Birthday, volume 3838 of Lecture Notes in Computer Science, pp. 173–197 (2005)

[98] Zantema, H.: Termination of string rewriting proved automatically. Journal of Automated Reasoning 34(2), 105–139 (2005)

[99] Zantema, H., Waldmann, J.: Termination by quasi-periodic interpretations. In: Proc. of the 18th International Conference on Rewriting Techniques and Applications (RTA 2007), volume 4533 of Lecture Notes in Computer Science, pp. 404–418 (2007)

[100] Zantema, H.: Termination. In: TeReSe (ed.) Term Rewriting Systems. Cambridge University Press, Cambridge, 181–259 (2003)

Appendix

Appendix A

T$_T$T$_2$

All (non-)termination methods presented in this thesis have successfully been implemented into the Tyrolean Termination Tool 2 (T$_T$T$_2$) available from

`http://cl-informatik.uibk.ac.at/software/ttt2/`.

T$_T$T$_2$ is open source (under LGPL version 3)[1] and written in OCaml. It interfaces the SAT solver MiniSat [20], the PB solver MiniSat+ [19], and the SMT solver Yices [18]. For a full system description please consider [56]. This chapter explains the syntax (Section A.1) and the semantics (Section A.2) of the strategy language which allows to flexibly configure T$_T$T$_2$. Section A.3 then explains how to call the tool on user-defined strategies while Section A.4 presents the strategies which have been employed for T$_T$T$_2$ to produce the experimental results listed in the tables within this thesis.

A.1 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 specifier is '`[`f`]`' (also 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

$$s ::= m \mid \texttt{(}s\texttt{)} \mid s\texttt{;}s \mid s\texttt{|}s \mid s\texttt{||}s \mid s\texttt{?} \mid s\texttt{+} \mid s\texttt{*} \mid s\texttt{[}f\texttt{]}$$

where m denotes any available method of T$_T$T$_2$ (possibly followed by some flags). The best way to get a complete and up-to-date list of all methods supported by T$_T$T$_2$ is to ask the tool itself. The command `./ttt2 --processors` returns a list of all processors available. A processor can also comment its parameters, i.e., `./ttt2 -s 'matrix -h' <file>` explains all flags the matrix method may take.

In order to avoid unnecessary parentheses, the following precedence is used: `?`, `+`, `*`, `[`f`]` $>$ `;` $>$ `|`, `||`.

[1] GNU Lesser General Public License, see `http://www.gnu.org/licenses/lgpl.html`.

A.2 Semantics

In the remainder of this section we use the notion *termination problem* to denote a TRS, DP problem, or relative termination problem. We call a termination problem *terminating* if the underlying TRS (DP problem, relative 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 always is `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, i.e., 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 random whose result is taken.

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, i.e., 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.

At last we explain the specifier '`[`f`]`' which 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`]` succeeds and returns the termination problem that remains after applying s to P.

Most strategy operators are demonstrated in the following example.

Example A.1. Consider the following strategy:

```
(var | uncurry?;poly -ib 2 -ob 4*;
       dp;edg;sccs;(matrix -dim 2 -dp[1.5] || kbo -dp)*)[5]
```

To (dis-)prove termination of a TRS $\mathcal{R}$, T$_{\text{T}}$T$_2$ performs the following steps. First the method `var` (a test if there exists a rewrite rule where the right-hand side contains a variable that is not present in its left-hand side) is applied. On success of this method non-termination is reported and the tool exits. If `var` fails then uncurrying is tried next. Since this method works only 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. Finally, after computing the dependency pairs, the estimated dependency graph, and the SCCs in the graph, T$_{\text{T}}$T$_2$ tries to prove finiteness of the current DP problems, by applying the strategy `(matrix -dim 2 -dp[1.5] || kbo -dp)` recursively. This strategy searches (in parallel) for compatible matrix interpretations (of dimension two) and KBOs. The flag `-dp` indicates that the respective method is applied in the DP setting (where strict monotonicity is not needed).

The specifier '`[5]`' ensures that T$_{\text{T}}$T$_2$ runs for at most five seconds. In addition we limit the time the matrix technique spends on a single DP problem to at most 1.5 seconds.

A.3 Specification and Configuration

The flag `--strategy` (or short `-s`) tells T$_{\text{T}}$T$_2$ to use the specified strategy. If the `-s` flag is omitted a predefined strategy is used (for details execute `./ttt2 --help`). The tool does not check if the strategy is sound, e.g., processors are applied in correct order. Calling T$_{\text{T}}$T$_2$ in a shell with the strategy from Example A.1, i.e., `./ttt2 -s '(var | uncurry?; ...)[5]' <file>` is already a bit inconvenient. Hence T$_{\text{T}}$T$_2$ supports to specify a configuration file which allows to abbreviate and connect different strategies. By convention strategy abbreviations are written in capital letters. To tell T$_{\text{T}}$T$_2$ which configuration file it should use, the flag `--conf` (or the short form `-c`) followed by the file name has to be set.

Example A.2. To call T$_{\text{T}}$T$_2$ using the strategy from Example A.1 we write a configuration file `ttt2.conf` containing the following lines:

```
PRE = uncurry?;poly -ib 2 -ob 4*
PARALLEL = (matrix -dim 2 -dp[1.5] || kbo -dp)
AUTO = (var | PRE;dp;edg;sccs;PARALLEL*)[5]
```

Abbreviations are not implicitly surrounded by parentheses since this allows more freedom in abbreviating expressions. To specify that the strategy `AUTO` from the configuration file `ttt2.conf` should be used amounts to the following command: `./ttt2 -c ttt2.conf -s AUTO <file>`.

A.4 TTT2 Strategies

This section lists the strategies employed for TTT2 to produce the experimental data shown in the various tables. Some clarifications are in order. For any strategy *s* below, actually `./ttt2 -s 'var | s' <file>` is executed. Here `var` filters out systems that are not TRSs (cf. Example A.1). Only the essential parts are listed in the table entries below. Replacing the □ in the corresponding strategy skeleton yields the exact strategy.

Strategies for Chapter 2

For Table A.1 the strategy skeleton `kbo` □ was used.

Table A.1: Strategies for Table 2.1

method(#bits)	strict precedence	quasi-precedence
sat/pbc(2)	`-sat/-pbc -ib 2`	`-quasi -sat/-pbc -ib 2`
sat/pbc(3)	`-sat/-pbc -ib 3`	`-quasi -sat/-pbc -ib 3`
sat/pbc(4)	`-sat/-pbc -ib 4`	`-quasi -sat/-pbc -ib 4`
sat/pbc(10)	`-sat/-pbc -ib 10`	`-quasi -sat/-pbc -ib 10`
smt_i	`-smt`	`-quasi -smt`
smt_r	`-smt -real`	`-quasi -smt -real`

Table A.2 employs the skeleton `kbo` □.

Table A.2: Strategies for Table 2.2

method(#bits)	strict precedence	quasi-precedence
sat/pbc(3)	`-sat/-pbc -ib 3`	`-quasi -sat/-pbc -ib 3`
sat/pbc(4)	`-sat/-pbc -ib 4`	`-quasi -sat/-pbc -ib 4`
sat/pbc(6)	`-sat/-pbc -ib 6`	`-quasi -sat/-pbc -ib 6`
sat/pbc(8)	`-sat/-pbc -ib 8`	`-quasi -sat/-pbc -ib 8`
smt_i	`-smt`	`-quasi -smt`
smt_r	`-smt -real`	`-quasi -smt -real`

Table A.3 adopts the skeleton `dp;edg;(sccs;ur;kbo -dp -ur` □`)*`.

Table A.3: Strategies for Table 2.3

method(#bits)	TRSs	SRSs
sat(2)	`-sat -ib 2`	`-sat -ib 2`
sat(3)	`-sat -ib 3`	`-sat -ib 3`
sat(4)	`-sat -ib 4`	`-sat -ib 4`
sat(5)	`-sat -ib 5`	`-sat -ib 5`
sat(6)	`-sat -ib 6`	`-sat -ib 6`
sat(10)	`-sat -ib 10`	`-sat -ib 10`
smt_i	`-smt`	`-smt`
smt_r	`-smt -real`	`-smt -real`

Strategies for Chapter 3

The strategy skeleton `dp;edg;(sccs;ur;matrix -dp -ur -ib 2 -ob 3 □)*` was used for Table A.4.

Table A.4: Strategies for Table 3.2

(a) TRSs

method	
direct_n	`-incn`
direct_e	`-ince`
a	`-inca`
b	`-incb`
compress	`-incc`

(b) SRSs

method	
direct_n	`-incn`
direct_e	`-ince`
a	`-inca`
b	`-incb`
compress	`-incc`

Strategies for Chapter 4

The strategy skeleton `dp;edg;(sccs;ur;matrix -dp -ur -dim □)*` was applied for Tables A.5 and A.6. We omit the strategies for 2×2 due to the similarity with the other dimensions. The exact values for `-ib` and `-ob` are obtained by subtracting one from the table entries in the columns labeled 1×1.

Table A.5: Strategies for Table 4.1

	1×1	3×3
$\mathbb{N}$	`1 -ib 4 -ob 5`	`3 -ib 2 -ob 3`
$\mathbb{Q}$	`1 -ib 4 -ob 5 -rat 2`	`3 -ib 2 -ob 3 -rat 2`
$\mathbb{Q}_1$	`1 -ib 4 -ob 5 -rat 2 -db 0`	`3 -ib 2 -ob 3 -rat 2 -db 0`
$\mathbb{Q}_2$	`1 -ib 4 -ob 5 -rat 2 -db 1`	`3 -ib 2 -ob 3 -rat 2 -db 1`
$\mathbb{R}$	`1 -ib 2 -ob 3 -real`	`3 -ib 1 -ob 2 -real`

Table A.6: Strategies for Table 4.2

	1×1	3×3
$\mathbb{N}$	`1 -ib 5 -ob 6`	`3 -ib 3 -ob 4`
$\mathbb{Q}$	`1 -ib 5 -ob 6 -rat 2`	`3 -ib 3 -ob 4 -rat 2`
$\mathbb{Q}_1$	`1 -ib 5 -ob 6 -rat 2 -db 0`	`3 -ib 3 -ob 4 -rat 2 -db 0`
$\mathbb{Q}_2$	`1 -ib 5 -ob 6 -rat 2 -db 1`	`3 -ib 3 -ob 4 -rat 2 -db 1`
$\mathbb{R}$	`1 -ib 3 -ob 4 -real`	`3 -ib 2 -ob 3 -real`

Strategies for Chapter 5

For Table 5.2 the strategy from Listing A.1 (TRSs) and the configuration file presented in Listing A.2 (SRSs) have been used.

Listing A.1: Finding loops for TRSs

```
var | con | poly -ib 3 -ob 4[10]*;( \
  unfold || unfold -fwd || unfold -bwd || \
   dp;edg;(sccs;ur;(poly -dp -ur -ib 2 -ob 4))*)
```

Listing A.2: Finding loops for SRSs

```
PRE = (\
 matrix -dim 1 -ib 3 -ob 5 || \
 matrix -dim 2 -ib 3 -ob 3 || \
 matrix -dim 3 -ib 2 -ob 3 || \
 matrix -dim 4 -ib 2 -ob 2 || \
 arctic -dim 1 -ib 3 -ob 4 || \
 arctic -dim 2 -ib 2 -ob 2 || \
 arctic -dim 3 -ib 1 -ob 2)
SIMP = (\
 matrix -dp -ur -dim 1 -ib 3 -ob 6[2] || \
 matrix -dp -ur -dim 2 -ib 2 -ob 3[2] || \
 matrix -dp -ur -dim 3 -ib 2 -ob 2[2] || \
 matrix -dp -ur -dim 4 -ib 1 -ob 2[5] || \
 arctic -dp -ur -dim 2 -ib 2 -ob 2[2] || \
 arctic -dp -ur -dim 3 -ib 2 -ob 2[3] || \
 arctic -dp -ur -dim 4 -ib 1 -ob 2[5])
NONTERM = (\
 loop -dp -sat -r  4 -c  5 || \
 loop -dp -sat -r  6 -c 25 || \
 loop -dp -sat -r 12 -c 12 || \
 loop -dp -sat -r 15 -c 12 || \
 loop -dp -sat -r 15 -c 15 || \
 loop -dp -sat -r 15 -c 18 || \
 loop -dp -sat -r 18 -c 15 || \
 loop -dp -sat -r 18 -c 18 || \
 loop -dp -sat -r 15 -c 20 || \
 loop -dp -sat -r 20 -c 15 || \
 loop -dp -sat -r 20 -c 20 || \
 loop -dp -sat -r 15 -c 22 || \
 loop -dp -sat -r 22 -c 15 || \
 loop -dp -sat -r 25 -c 15 || \
 loop -dp -sat -r 22 -c 22 || \
 loop -dp -sat -r 20 -c 25 || \
 loop -dp -sat -r 25 -c 20)
LOOPSAT = \
 (PRE[2]*;dp;edg;(sccs;ur;(SIMP || NONTERM))*)
```

Index