FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Extending VMTL by a More Flexible Control and New Methods

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Sebastian Graczoll
Matrikelnummer 0425773

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Inf. Dr.rer.nat. Bernhard Gramlich

Wien, 22.11.2011

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Extending VMTL by a More Flexible Control and New Methods

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Sebastian Graczoll
Registration Number 0425773

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Inf. Dr.rer.nat. Bernhard Gramlich

Vienna, 22.11.2011

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Sebastian Graczoll
Traminergasse 8, 3434 Tulbing

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

The completion of this thesis would not have been possible without the support of many people.

First, I want to thank my advisor Prof. Dr. Bernhard Gramlich for giving me the opportunity to work on this very interesting project and to be part of the VMTL development team. He always found time for extensive and fruitful meetings and discussions about my thesis in which most of the ideas that have been realized originated. His great experience in the field of term rewriting and scientific working have led to many invaluable suggestions, hints and remarks, both technically and stylistically, that greatly improved and shaped the final outcome of this thesis.

Of course none of this work would have been possible without Dr. Felix Schernhammer, who started the VMTL project and set up a solid, well documented and extensible framework which allowed me to integrate my extensions without any problems. I am very grateful for his continued support. Whenever I had any questions about the VMTL code base, he always helped me out quickly and competently. Through his help and experience, the extensive tests of the extensions could be automated, which highly simplified the process.

I also want to thank Florian Schweikert, the system administrator of the institute, for providing me with a dedicated server for running the tests of the new VMTL versions and for his support in configuring the server to my requirements.

I would also like to thank my employer, the Engineering Geodesy Group of the Institute of Geodesy and Geophysics at the TU Wien, especially, the head of the group, Prof. Dr. Andreas Wieser, for the high degree of flexibility offered, concerning my work hours, which allowed me to work efficiently on my thesis.

Finally, I want to thank my family and my friends for their ongoing support throughout the course of writing this thesis.

Abstract

VMTL (Vienna Modular Termination Laboratory) is a tool for automated termination proof search for term rewriting systems (TRSs). It is based on the dependency pair (DP) framework. In this setting an initially given TRS is first transformed into a so-called DP problem which is then successively simplified using DP-processors until (hopefully) the problem becomes trivial, which means that termination of the initial TRS has been proved.

Once the DP method has been started, it is not possible any more to go back from DP problems to corresponding TRSs. Therefore, it is desirable to have strong and flexible preprocessing mechanisms at hand that simplify the original TRS as much as possible, in a way such that (non-)termination is preserved, before the DP analysis is started.

In this master's thesis, we design and implement a preprocessing framework for VMTL, that allows for a modular integration of direct termination proof methods.

Furthermore, two new approaches for proving termination of TRSs are added, namely Knuth-Bendix Orderings (KBO) and Semantic Labeling. Both techniques are incorporated as direct methods and as DP-processors.

In many cases, the proof search of direct methods and DP processors can be modelled efficiently as a SAT or SMT problem. In order to ease the generation and representation of such problems, we develop a new SAT/SMT solving toolkit for use within VMTL.

Kurzfassung

VMTL (Vienna Modular Termination Laboratory) ist ein Tool zur automatischen Suche von Terminationsbeweisen für Termersetzungssysteme (TRSs). Die Beweissuche in VMTL basiert auf dem Dependency Pair (DP) Framework. Dabei wird das auf Termination zu untersuchende TRS zunächst in das zugehörige sogenannte DP-Problem transformiert, welches dann durch Anwendung von DP-Prozessoren sukzessive vereinfacht wird, bis das Problem (im besten Fall) trivial wird, wodurch dann Termination des ursprünglichen TRS bewiesen ist.

Ein Nachteil der DP Methode ist, dass es im Allgemeinen nicht möglich ist, von DP-Problemen wieder zu den entsprechenden TRSs zurückzukehren. Aus diesem Grund ist es wünschenswert, ein flexibles Framework für Preprocessing zur Verfügung zu haben, um bereits vor der DP-Analyse Vereinfachungen durchführen zu können, die die (Nicht-)Terminationseigenschaften des zu untersuchenden TRS bewahren.

Eines der Ziele dieser Arbeit ist der Entwurf und die Implementierung eines solchen Preprocessing Framework auf Basis klassischer (direkter) Terminationsbeweismethoden.

Weiters werden im Zuge dieser Arbeit zwei neue Ansätze für Terminationsbeweismethoden implementiert: Knuth-Bendix Orderings (KBO) sowie Semantic Labeling. Beide Techniken werden sowohl als direkte Methoden wie auch als DP-Prozessoren umgesetzt.

Die Beweissuche vieler Methoden und Prozessoren lässt sich effizient als SAT- oder SMT-Problem formulieren. In dieser Arbeit entwickeln wir ein neues SAT/SMT-solving Toolkit, welches innerhalb von VMTL genutzt werden kann, um SAT/SMT-Aufgaben zu formulieren und zu lösen.

Contents

1 Overview **1**
- 1.1 Goals of this Thesis . . . 1
- 1.2 Outline of this Document . . . 3

2 Introduction **5**
- 2.1 Term Rewriting . . . 5
  - Termination of Term Rewriting Systems . . . 6
- 2.2 Context-Sensitive Term Rewriting . . . 7
- 2.3 Conditional Term Rewriting . . . 8

3 Preliminaries **9**
- 3.1 Basics . . . 9
- 3.2 Term Rewriting . . . 12
  - Termination . . . 13
  - The Dependency Pair Framework . . . 14
- 3.3 Context-Sensitive Rewriting . . . 15
  - Termination of CS-TRSs . . . 16
  - The Context-Sensitive Dependency Pair Framework . . . 16
- 3.4 Conditional Rewriting . . . 19

4 VMTL **21**
- 4.1 VMTL . . . 21
  - Strategies for the DP Analysis . . . 22
  - Context-Sensitive Rewriting . . . 28
  - Conditional Rewriting . . . 28
  - TRS-to-TRS Transformations . . . 28
  - Output Processing . . . 28
- 4.2 Extensions of VMTL . . . 29
  - Direct Methods . . . 29
  - Extensions of the Strategy Specification . . . 30
  - Strategy Trees for Preprocessing . . . 31
  - Executing Strategy Trees for Preprocessing . . . 31
- 4.3 The VMTL Source Code . . . 35

DeepVisDS . . . 35
DeepVis . . . 36
DeepVisCLI . . . 36

5 The VMTL Sat Solving Facility **37**
5.1 Introduction . . . 37
5.2 The Specification Logics . . . 39
Syntax . . . 40
Semantics . . . 41
Satisfiability of **T**-Logic Formulas . . . 43
5.3 Built-In Theories of the VSSF . . . 43
BitVectors . . . 44
Integers . . . 45
Reals . . . 46
5.4 Overview of the VSSF . . . 46
Problem Specification . . . 47
Solving . . . 56
Models . . . 60
5.5 The Factory Classes . . . 62
5.6 Using the VSSF . . . 64

6 Semantic Labeling **69**
6.1 Semantic Labeling . . . 69
Semantics of a TRS, Algebras . . . 69
Semantic Labeling . . . 71
Semantic Labeling for Context-Sensitive Rewriting . . . 72
Semantic Labeling as a CS-DP processor . . . 74
6.2 Root-Labeling . . . 76
The Direct Method . . . 76
The DP Processor . . . 77
6.3 Generalizing Root-Labeling to the Context-Sensitive Case . . . 79
6.4 Generalizing the Root-Labeling Processor to the Context-Sensitive Case . . . 81
6.5 Implementation for VMTL . . . 85
The Direct Method . . . 86
The (CS-)DP Processor . . . 86

7 Knuth-Bendix Order **89**
7.1 Knuth-Bendix Order . . . 89
7.2 Implementation as a Direct Method . . . 90
Specifying the Problem Formula . . . 91
Solving the Problem Formula via SAT/SMT Solving . . . 93
7.3 Implementation as a (CS-)DP Processor . . . 93
The Reduction Pair Processor . . . 93
The Context-Sensitive Reduction Pair Processor . . . 97

A SAT/SMT Encoding . . . 99
7.4 Implementation for VMTL . . . 104
The Direct Method . . . 104
The (CS-)DP Processor . . . 106

8 Tests and Benchmarks **109**
8.1 Knuth-Bendix-Order . . . 109
The Direct Method . . . 109
The DP Processor . . . 111
8.2 Root-Labeling . . . 113
8.3 A New Default Strategy . . . 115
VMTL – Original Version . . . 115
VMTL – Extended Version . . . 116
Comparison with Other Tools . . . 117

9 Conclusions **121**
9.1 Summary . . . 121
9.2 Related Work . . . 123
9.3 Further Development . . . 124

A Strategy **125**
A.1 Default Strategy . . . 125
A.2 XML Schema Definitions . . . 129
XSD Template . . . 129
Example of Compiled XSD . . . 130

B Example Outputs **135**
B.1 KBO . . . 135
The Direct Method . . . 135
The (CS-)DP Processor . . . 137
B.2 Root-Labeling . . . 139
The Direct Method . . . 139
The (CS-)DP Processor . . . 141

Bibliography **143**

Index **149**

CHAPTER 1

Overview

This chapter provides a short overview of the tasks and goals of this thesis (Section 1.1) and discusses how they are approached. In the first section of this chapter, we use some terms that are not yet defined. These terms will be clarified in the following chapters in detail. The index also provides a guide to the definitions of the most important terms. Section 1.2 gives an overview of the remaining chapters in this thesis and describes how they are related.

1.1 Goals of this Thesis

The topic of this master's thesis is an extension of VMTL by a more flexible control and new methods. VMTL (Vienna Modular Termination Laboratory, [SG09]) is a termination prover for term rewriting systems. It was developed at the Institut für Computersprachen at the TU Wien by Felix Schernhammer.

The main goals of this thesis were:

- Adding support for direct proof methods to VMTL.
- Reworking the strategy language in order to support the new methods.
- Designing and implementing an unified SAT/SMT solving toolkit for use within VMTL.
- Implementing the proof technique of semantic labeling [Zan95] (more specifically root-labeling [SM08, ST10]) for VMTL.
- Implementing the proof technique of Knuth-Bendix orders [KB70] for VMTL.
- Adding the new techniques as direct methods as well as processors in the dependency pair framework [AG00, GTSK05].
- Testing and evaluation of the new methods.

The first step will be the design and implementation of the new framework for direct proof methods. Here, it is important that context-sensitive rewriting is supported, and that a great degree of modularity is offered in order to do justice to the term *modularity* in the name of VMTL. To this end, an approach similar to the DP processors in the DP framework is used. In order to allow the users of VMTL to use the new direct methods, it is also necessary to revise the strategy specification language for VMTL. In this process, a new strategy language based on XML will be designed and implemented. The use of XML seems very handy in the new strategy language, as powerful XML parsers for Java (VMTL is implemented in Java) exist, and XML is very well suited for specifying tree-like structures, as they are needed in the strategy specification. Furthermore, the use of XSD (XML schema definition) allows to check strategy files for syntactical and basic logical errors and to notify the user in a user-friendly, human-readable way.

Another important part of this thesis is the design and implementation of a SAT/SMT solving toolkit for use in VMTL. This was necessary, because previously there was no unified way to use SAT solving in VMTL. Therefore, processors requiring SAT solving have to reimplement functionality such as DIMACS [Dim93] input generation and output parsing, starting of the SAT solving process, constraint generation etc. all over again, every time. Thus it was desirable to have a one-stop toolkit that encapsulates all this functionality and in addition offers strong constraint generation tools. Furthermore, it should be possible to use other back-ends like SMT solving instead of SAT solving in a (largely) transparent way. If an SMT back-end is used, special features of SMT should be utilized as well as possible, to make computations efficient.

The first practical use of the newly developed SAT/SMT solving toolkit will be the implementation of the Knuth-Bendix order [KB70] as a direct proof method and as a (CS-)DP processor. This implementation will mainly rely on the SAT encoding presented in [ZHM09].

Another technique that will be added is called root-labeling [SM08, ST10], a special case of semantic labeling [Zan95]. Again, the implementation will follow the approach presented in [SM08] resp. [ST10]. For the implementation of the context-sensitive versions of the direct proof method and the CS-DP processor, we will generalize the existing techniques.

All in all, the contributions of this thesis are the following:

- VMTL is improved and made more flexible.
- A new versatile SAT/SMT solving toolkit for VMTL is designed and implemented.
- The KBO processor from [ZHM09] is extended to support the CS-DP framework.
- Some of the results from [Zan95] and [SM08] (concerning semantic labeling and root-labeling) are generalized to the context-sensitive case.
- Tests and evaluation of the new techniques.

1.2 Outline of this Document

The thesis is structured in nine chapters. Each chapter begins with an overview of the content covered in the chapter and an outline of the sections of the chapter.

The following list summarizes the contents of the chapters:

Chapter 1 - Overview
Contains a summary of the goals of this thesis and gives an overview of the structure of the written part of this thesis.

Chapter 2 - Introduction
Gives a very brief, informal introduction to the formalism of term rewriting and termination of term rewriting systems. Additionally, various extensions to term rewriting are presented.

Chapter 3 - Preliminaries
Contains a more in-depth, formal introduction of the concepts presented in Chapter 2. This chapter also contains the most important definitions and conventions used in the subsequent chapters and provides the basis for the unified notation used throughout the thesis.

Chapter 4 - VMTL Prerequisites: Chapter 3
In this chapter we discuss some of the most important features of VMTL, and specify which kind of extensions are implemented as part of this thesis. Here, the new direct proof methods and the strategy extension are discussed in detail.

Chapter 5 - The VMTL Sat Solving Facility
This chapter provides an introduction to the VMTL SAT solving facility (VSSF).

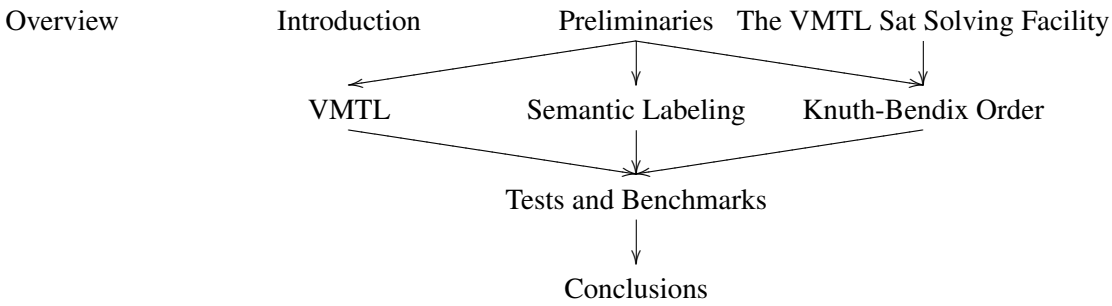
Chapter 6 - Semantic Labeling Prerequisites: Chapter 3
Introduces semantic labeling [Zan95] and, as special case thereof, root-labeling [SM08, ST10]. The existing results are generalized to the context-sensitive case, as basis for the implementation in VMTL.

Chapter 7 - Knuth-Bendix Order Prerequisites: Chapters 3, 5
Introduces the Knuth-Bendix order [KB70] and the SAT/SMT encoding of the direct method and DP processor [ZHM09]. The DP processor is extended to the context-sensitive case for implementation in VMTL.

Chapter 8 - Tests and Benchmarks Prerequisites: Chapters 4, 6, 7
Describes the tests of the new methods and processors and presents a new default strategy for VMTL.

Chapter 9 - Conclusions Prerequisites: Chapter 8
Summarizes the work of the thesis and provides references to related material.

The following graph depicts the dependency of the chapters:



CHAPTER 2

Introduction

This chapter provides some informal overview of the most important concepts used in this thesis. We start with a very brief high-level description of term rewriting and termination. After that we give a short informal introduction to the dependency pair framework. Next, we consider some more advanced forms of term rewriting like context-sensitive rewriting and conditional term rewriting.

2.1 Term Rewriting

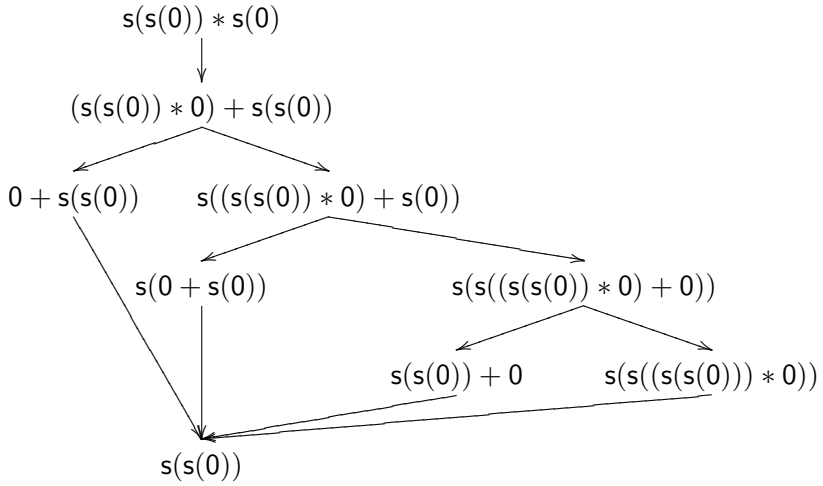
Term rewriting is a versatile formalism with applications in different fields of computer science. Basically, term rewriting describes in a formal way how terms can be replaced by other terms while maintaining equality in some sense. In term rewriting, the rewrite steps are oriented. Usually a rewrite step *simplifies* terms, thus rewrite steps are also called *reduction* steps. Term rewriting has a close relation to functional programming. Therefore, it is very well suited for reasoning about functional programs and algorithms in general. Other areas where term rewriting is used are for example formal verification of software and model checking, automatic theorem proving, logic, algebra etc...

A term rewriting system (TRS) consists of a number of rules, where a rule has a left-hand-side (lhs) and a right-hand-side (rhs). Both the lhs and rhs are so-called terms. A term can contain variables, which can be replaced by other terms. Whenever a (instantiation of a) lhs of a rule in a TRS matches a subterm of some term, this rule can be applied, and the matching subterm can be replaced by the (instantiation of the) rhs of the rule.

Example 2.1.1.

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathsf{0}{+}x & \to & x & \mathsf{s}(x){+}y & \to & \mathsf{s}(x{+}y) \\ x{+}\mathsf{0} & \to & x & x{+}\mathsf{s}(y) & \to & \mathsf{s}(x{+}y) \\ x{*}\mathsf{0} & \to & \mathsf{0} & x{*}\mathsf{s}(y) & \to & (x{*}y){+}x \end{array}$$

This term rewriting system consisting of six rewrite rules can be used to perform addition and multiplication on natural numbers. Here a number is expressed by means of repeated applications of the successor function s on the constant 0. For example the number 2 is represented as $\mathsf{s}(\mathsf{s}(0))$. The term representing the multiplication of 2 and 1 would therefore be $\mathsf{s}(\mathsf{s}(0)) * \mathsf{s}(0)$. Now, the third rule on the right side can be applied, resulting in the term $(\mathsf{s}(\mathsf{s}(0)) * 0) + \mathsf{s}(\mathsf{s}(0))$. At this point, there are two ways to proceed. Either the third rule on the left side is used or the second rule on the right side is used. The following tree depicts some possible ways of reaching a normal form (i.e. a term that cannot be reduced any further).



As expected, all paths eventually lead to the same normal form $\mathsf{s}(\mathsf{s}(0))$.

Term rewriting allows elegant reasoning over questions asked about systems specified within the formalism. For example, one could ask about certain properties such as confluence and termination of the rewrite relation induced by a term rewriting system. Over the years, much theory has been worked out, that allows formal justification of answers to these questions.

Termination of Term Rewriting Systems

One of the most interesting and most actively explored questions is, whether a certain rewrite system is terminating or not. A term rewriting system is called terminating (or strongly normalizing, SN), if there is no term which permits an infinite rewrite sequence. In general, this question is undecidable (because term rewriting is Turing-complete) [BN98]. However, very powerful techniques have been developed, so today a broad class of term rewriting systems can be proven terminating (or non-terminating). The most common way of formally proving termination of term rewriting systems is by finding an order with certain properties (called reduction order, see Chapter 3.2) on terms, such that for each rule in the TRS, the lhs is greater than the rhs.

However, in recent years the emphasis of proving termination has shifted from pure formal considerations to considerations about how automated proving of termination becomes feasi-

ble. One of the most successful approaches, which is used in almost all competitive termination provers today (e.g. AProVE [GTSKF04], TTT [KSZM09], VMTL [SG09]), is the so-called dependency pair (DP) framework ([AG00, GTSK05], for a short formal introduction see chapter 3.2). The basic idea of the DP framework is to exploit certain properties of rewrite sequences starting at minimal non-terminating terms (i.e. terms where all proper subterms are terminating). By capturing the recursive rewrite structure of the rules (called dependency pairs, which together with the original rules themselves form a so-called DP problem), it becomes possible to reason about such sequences. The DP framework is called a framework, because it allows the definition of so called DP processors that operate on DP problems. DP processors can be applied consecutively, successively simplifying the DP problems, until at a certain point termination becomes immediate. This modularity makes the DP framework very well suited for automated proof search because many different DP processors were developed, having different characteristics, some of which allow partitioning of DP problems into smaller DP problems (e.g. the dependency graph processor [GTSK05]) and others making progress in simplifying DP problems by removing some dependency pairs and/or rewrite rules (e.g. the reduction pair processor [GTSK05]).

2.2 Context-Sensitive Term Rewriting

Context-sensitive rewriting [Luc98] is a generalization of term rewriting, where it is possible to restrict the induced rewrite relation by defining at which positions in a term rewrite steps are allowed to be applied. This makes it very useful for modelling features of functional programming.

Example 2.2.1. Consider the following context-sensitive rewrite system (CS-TRS):

$$\begin{array}{rcl} \mathsf{if}(\mathsf{true}, x, y) & \to & x \\ \mathsf{s}(x) > \mathsf{s}(y) & \to & x > y \\ \mathsf{s}(x) > \mathsf{0} & \to & \mathsf{true} \end{array} \qquad \begin{array}{rcl} \mathsf{if}(\mathsf{false}, x, y) & \to & y \\ \mathsf{0} > y & \to & \mathsf{false} \end{array}$$

Let reduction steps be allowed everywhere, except at the second and third arguments of if terms. So for example in the term $\mathsf{if}(\mathsf{s}(\mathsf{s}(\mathsf{0})) > \mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}) > \mathsf{0}, \mathsf{0})$, reduction steps are only allowed at the first argument of if until eventually the term $\mathsf{if}(\mathsf{true}, \mathsf{s}(\mathsf{0}) > \mathsf{0}, \mathsf{0})$ is reached. Then the first rule can be applied to obtain $\mathsf{s}(\mathsf{0}) > \mathsf{0}$. Now, this term can be further reduced to true.

There are different approaches to proving termination of CS-TRSs. One possibility is to transform CS-TRSs into TRSs and then to use standard techniques for TRSs. The most simple transformation is to just ignore the position constraints in the CS-TRS. This method is sound but of course highly incomplete (i.e. many CS-TRSs that are terminating become non-terminating by this transformation). It was shown, that there are indeed sound and complete transformations of CS-TRSs to TRSs [GM99], thus theoretically all methods that work for TRSs also can be applied for CS-TRSs. However, especially for automated termination proving, in practice it is more efficient to use new techniques or generalizations of existing techniques that are crafted specially for termination proofs of CS-TRSs.

One of the methods that have been successfully adopted for context-sensitive rewriting is the DP framework (the context-sensitive version being called CS-DP framework [AGL06, AEF$^+$08]). Most existing DP processors can be generalized to work in this new framework very easily, so today already quite a big number of CS-DP processors exists.

2.3 Conditional Term Rewriting

Another very useful variant of term rewriting is called *conditional term rewriting* ([Kap84, BK86] cf. [Ohl02]). Here, applicability of rewrite rules can depend on one or more conditions. Conditions can be expressed in different ways. A common way is to express conditions in the formalism of term rewriting, by requiring that some term can be reduced to some other term (such a conditional term rewriting system (CTRS) is called *oriented* CTRS).

Example 2.3.1. Consider the following CTRS:

$$\begin{array}{rcl} \mathsf{even}(\mathsf{s}(\mathsf{s}(x))) & \to & \mathsf{even}(x) \\ \mathsf{even}(\mathsf{s}(\mathsf{0})) & \to & \mathsf{false} \\ \mathsf{even}(\mathsf{0}) & \to & \mathsf{true} \\ \mathsf{odd}(x) & \to & \mathsf{true} \Leftarrow \mathsf{even}(x) \to^* \mathsf{false} \\ \mathsf{odd}(x) & \to & \mathsf{false} \Leftarrow \mathsf{even}(x) \to^* \mathsf{true} \end{array}$$

Computation of $\mathsf{odd}(x)$ is defined only through the computation of $\mathsf{even}(x)$. The last two rules are conditional. The rule $\mathsf{odd}(x) \to \mathsf{true}$ may only be applied, if $\mathsf{even}(x)$ can be reduced to false.

Conditional term rewriting usually requires a different notion of termination than that for term rewriting or context-sensitive term rewriting. This is because it may be undecidable, whether a condition holds or not. To this end, the notion of *effective termination* [Ohl02] was introduced, where in addition to termination of the rewrite relation it is required that the rewrite relation is decidable. However, it was shown that in practice this is still not sufficient because, even though decidable, implementations trying to check whether a condition holds may loop. Therefore, [LMM05] defines the even stronger notion of *operational termination* which is what VMTL proves for CTRSs.

The proof of operational termination can be done in a two-step process. First the CTRS is transformed into a CS-TRS by means of *context-sensitive unravelling* [SG10, Sch11]. It is shown in [SG10, Sch11], that termination of the resulting CS-TRS implies operational termination of the original CTRS.

CHAPTER 3

Preliminaries

This chapter provides an overview and a reference for the most important definitions, notations and terminology used commonly throughout this thesis. A more in-depth introduction to term rewriting can be found in [BN98]. This chapter contains four sections: Section 3.1 contains the most important definitions that are the formal basis for the other sections. Section 3.2 gives a short introduction to term rewriting and some methods of proving termination. Section 3.3 gives an overview of context-sensitive rewriting and termination methods for context-sensitive rewriting. The last section gives a short introduction to conditional rewriting.

3.1 Basics

Throughout this thesis $X = \{x, y, z, \ldots\}$ denotes a fixed, countable infinite *set of variables*.

Definition 3.1.1 (Signature)**.** A *signature* is a set $\mathcal{F}$ containing function symbols, disjoint from X. The function $\mathtt{ar}\colon \mathcal{F} \mapsto \mathbb{N}_0$ assigns to each function symbol a fixed arity. The subset of $\mathcal{F}$ containing only the function symbols with arity n in $\mathcal{F}$ is denoted as $\mathcal{F}^{(n)}$ and is defined as $\{f \mid f \in \mathcal{F}, \mathtt{ar}(f) = n\}$.

Terms are built over function symbols from a signature and variables from X. The set of terms over a signature $\mathcal{F}$ and variables X is defined inductively as follows:

Definition 3.1.2 (Term)**.** Let $\mathcal{F}$ be a signature. The set of all *terms* over function symbols from $\mathcal{F}$ and variables from X is denoted as $\mathcal{T}(\mathcal{F}, X)$. The set $\mathcal{T}(\mathcal{F}, X)$ is the smallest set such that

- $X \cup \mathcal{F}^{(0)} \subseteq \mathcal{T}(\mathcal{F}, X)$
- $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, X)$ for all $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, X)$ and $f \in \mathcal{F}^{(n)} (n \geq 1)$

Definition 3.1.3 (Root Symbol)**.** The *root symbol* of a term is the function symbol or variable occurring outermost. The mapping $\mathtt{root}\colon \mathcal{T}(\mathcal{F}, X) \setminus X \mapsto \mathcal{F} \cup X$ is used to obtain the root

symbol of a term. For terms $t = f(t_1, \ldots, t_n)$, $\texttt{root}(t)$ is f for all $f \in \mathcal{F}^{(n)}$ and for $t = c$, $\texttt{root}(t)$ is c for all $c \in \mathcal{F}^{(0)}$. For variables, the root is the variable itself, i.e. $\texttt{root}(x) = x$ for all $x \in X$.

Definition 3.1.4 (Subterms, Immediate Subterms)**.** Let t be a term. The set $\texttt{Sub}(t)$ of all *subterms* of t is defined as the smallest set such that $t \in \texttt{Sub}(t)$ and for all terms $f(t_1, \ldots, t_n) \in \texttt{Sub}(t)$, $\texttt{Sub}(t_1) \subseteq \texttt{Sub}(t), \ldots, \texttt{Sub}(t_n) \subseteq \texttt{Sub}(t)$. The *immediate subterms* or *argument terms* of a term $f(t_1, \ldots, t_n)$ are the terms $t_1, \ldots, t_n$.

Definition 3.1.5 (Subterm Relation)**.** The *subterm relation* is a binary relation on terms denoted as $\trianglelefteq$, such that for two terms $s, t \in \mathcal{T}(\mathcal{F}, X)$, $s \trianglelefteq t$ if and only if $s \in \texttt{Sub}(t)$.

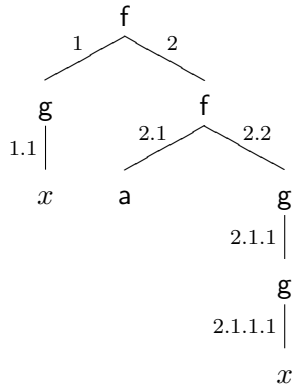
The function $|t|_s$ is used to count the number of occurrences of some term s in a term t.

Definition 3.1.6. Let t and s be terms in $\mathcal{T}(\mathcal{F}, X)$. The function $|\cdot|_\cdot : (\mathcal{T}(\mathcal{F}, X) \times \mathcal{T}(\mathcal{F}, X)) \mapsto \mathbb{N}_0$ is defined as follows.

$$|t|_s = \begin{cases} 1 & \text{if } t = s \\ \sum_{i=1}^{n} |t_i|_s & \text{if } t \neq s \text{ and } t = f(t_1, \ldots, t_n) \end{cases}$$

A term can be represented as a tree where each node is labeled by a function symbol or a variable and child nodes represent argument terms.

Example 3.1.7. For example, the term $t = \mathsf{f}(\mathsf{g}(x), \mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{g}(x))))$ corresponds to the following tree representation:



Each node in this tree represents a subterm of t. Each subterm of a term t has a *position* in t. A position is a (possibly empty) string of integers, describing the path in the tree which leads to the root symbol of some subterm.

Definition 3.1.8 (Positions)**.** Let $t \in \mathcal{T}(\mathcal{F}, X)$. The set $\texttt{Pos}(t)$ is the set containing all *subterm positions* in t. This set is constructed inductively as the smallest set $\texttt{Pos}(t)$ containing ϵ (called the *root position* of t) and if $t = f(t_1, \ldots, t_n)$ then

$$1\pi_1 \in \texttt{Pos}(t), \ldots, n\pi_n \in \texttt{Pos}(t) \text{ for all } \pi_1 \in \texttt{Pos}(t_1), \ldots, \pi_n \in \texttt{Pos}(t_n)$$

Notation. For better readability and unambiguity, we will usually use the symbol '.' to separate single numbers in a position string. For example, the position 123 will (depending on its supposed meaning) usually be written as either 123 or 1.23 or 12.3 or $1.2.3$.

Example 3.1.9. In the tree in Example 3.1.7, the edges are labeled with the positions of their descendant terms.

Definition 3.1.10 (Prefix Order, [BN98])**.** Let $t \in \mathcal{T}(\mathcal{F}, X)$ be a term and let $\pi \in \mathtt{Pos}(t)$ and $\pi' \in \mathtt{Pos}(t)$ be positions in t. The *prefix order* $\leq$ is defined as:

$$\pi' \leq \pi \text{ iff there exists } \pi'' \text{ such that } \pi'\pi'' = \pi$$

A position π' is *above* a position π if $\pi' \leq \pi$. π' is *strictly above* π if $\pi' < \pi$. The notions *below* and *strictly below* are defined analogously. If π and π' are incomparable with respect to $\leq$, we say that they are at *parallel* positions written as $\pi||\pi'$.

Positions can be used to extract subterms of a term. To this end, we will use the following operation:

Definition 3.1.11. Let $t \in \mathcal{T}(\mathcal{F}, X)$ be a term and $\pi \in \mathtt{Pos}(t)$ a position in t. $t|_\pi$ denotes the subterm at position π in t and is obtained as follows:

$$t|_\pi = \begin{cases} t & \text{if } \pi = \epsilon \\ t_i|_{\pi'} & \text{if } \pi = i\pi' \text{ and } t = f(t_1, \ldots, t_i, \ldots, t_n) \end{cases}$$

Definition 3.1.12 (Variable Positions, Non-Variable Positions)**.** The set of variable positions is defined as $\mathtt{VPos}(t) = \{\pi \mid \pi \in \mathtt{Pos}(t), t|_\pi \in X\}$ and contains all positions of subterms in t that are variables. The set $\mathtt{FPos}(t) = \mathtt{Pos}(t) \setminus \mathtt{VPos}(t)$ of non-variable positions contains all positions of subterms of t that are not variables.

Definition 3.1.13 (Variables of a term, Function symbols of a term)**.** Let $t \in \mathcal{T}(\mathcal{F}, X)$ be a term. The set $\mathtt{Vars}(t) = \{x \mid \pi \in \mathtt{VPos}(t), t|_\pi = x\}$ contains all variables occurring in t. A term that does not contain any variables (i.e. $\mathtt{Vars}(t) = \emptyset$) is called a *ground term* or *ground*. The set $\mathtt{Func}(t) = \{f \mid \pi \in \mathtt{FPos}(t), \mathtt{root}(t|_\pi) = f\}$ contains all function symbols occurring in t.

In a term, subterms can be replaced by other terms. This way, a new term is obtained. $t[s]_\pi$ denotes the term t, where the subterm at position π is replaced by the term s. Formally:

Definition 3.1.14. Let $t, s \in \mathcal{T}(\mathcal{F}, X)$ be terms and let $\pi \in \mathtt{Pos}(t)$ be a position in the term t. Then $t[s]_\pi$ is defined as:

$$t[s]_\pi = \begin{cases} s & \text{if } \pi = \epsilon \\ f(t_1, \ldots, t_i[s]_{\pi'}, \ldots, t_n) & \text{if } \pi = i\pi' \text{ and } t = f(t_1, \ldots, t_i, \ldots, t_n) \end{cases}$$

Contexts are special terms over a signature $\mathcal{F} \cup \{\Box\}$, where $\Box$ (hole) is a fresh constant symbol. A context contains $\Box$ at one or more positions. The hole-positions are intended to be replaced by some term when the context is used.

Definition 3.1.15 ($\mathcal{T}(\mathcal{F}, X)$-Substitution)**.** A *substitution* for a set of terms $\mathcal{T}(\mathcal{F}, X)$ (called $\mathcal{T}(\mathcal{F}, X)$-substitution or just substitution if the set of terms is clear from the context or does not matter) is a mapping $\sigma\colon X \mapsto \mathcal{T}(\mathcal{F}, X)$. A $\mathcal{T}(\mathcal{F}, X)$-substitution can be extended to a mapping σ' from terms to terms by homomorphic extension:

$$\sigma'(t) = \begin{cases} \sigma(x) & \text{if } t = x \in X \\ c & \text{if } t = c \in \mathcal{F}^{(0)} \\ f(\sigma'(t_1), \ldots, \sigma'(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \text{ with } f \in \mathcal{F}^{(n)} \text{ and } n \geq 1 \end{cases}$$

By abuse of notation, in the following when talking about a substitution σ, we always consider its homomorphic extension σ' but denote it as σ as well. The set $\texttt{Subst}(\mathcal{T}(\mathcal{F}, X))$ is the set containing all $\mathcal{T}(\mathcal{F}, X)$-substitutions.

Definition 3.1.16 (Instance)**.** A term $t' \in \mathcal{T}(\mathcal{F}, X)$ is called an *instance* of a term $t \in \mathcal{T}(\mathcal{F}, X)$, if there is some $\mathcal{T}(\mathcal{F}, X)$-substitution σ such that $\sigma(t) = t'$.

3.2 Term Rewriting

The formalism of term rewriting describes how terms can be rewritten into other terms by using rules that define the allowed rewrite steps.

Definition 3.2.1 (Rewrite Rule)**.** A *rewrite* rule (or just rule) is a pair (l, r) of terms l and r, usually denoted as $l \to r$. The term l is called the *left-hand-side* (lhs) of the rule and r is called the *right-hand-side* (rhs) respectively. A term t that is an instance of a lhs of a rule is called a *redex*.

Definition 3.2.2 (Term Rewriting System, TRS)**.** A *term rewriting system* (*TRS*) $\mathcal{R}$ is a pair $(\mathcal{F}, R)$ where $\mathcal{F}$ is a signature and R is a set of rules such that, for every rule $l \to r \in R$, $l, r \in \mathcal{T}(\mathcal{F}, X)$ and the following restrictions hold:

- $\texttt{Vars}(r) \subseteq \texttt{Vars}(l)$
- $l \notin X$

For better readability, we will write $l \to r \in \mathcal{R}$ when we really mean $l \to r \in R$ for a TRS $\mathcal{R} = (\mathcal{F}, R)$.

A function symbol $f \in \mathcal{F}$ is called *defined* in a TRS $\mathcal{R}$, if f is the root symbol of some left-hand-side of a rule in $\mathcal{R}$. A TRS where in each rule, no variable occurs more than once in the lhs is called *left-linear*.

Definition 3.2.3 (Closure under $\mathcal{F}$-Contexts, Closure under $\mathcal{T}(\mathcal{F}, X)$-substitutions)**.** Let $\circ$ be a binary relation on terms from $\mathcal{T}(\mathcal{F}, X)$. The relation $\circ$ is:

- *closed under $\mathcal{F}$-contexts* if and only if for every two terms t and s from $\mathcal{T}(\mathcal{F}, X)$, $t \circ s$ implies $C[t]_\pi \circ C[s]_\pi$ for all $C \in \mathcal{T}(\mathcal{F}, X)$ and $\pi \in \texttt{Pos}(C)$.

- *closed under $\mathcal{T}(\mathcal{F}, X)$-substitutions* if and only if for every two terms t and s from $\mathcal{T}(\mathcal{F}, X)$, $t \circ s$ implies $\sigma(t) \circ \sigma(s)$ for all $\mathcal{T}(\mathcal{F}, X)$-substitutions σ.

Definition 3.2.4 (Rewrite Relation)**.** A *rewrite relation* on terms from $\mathcal{T}(\mathcal{F}, X)$ is a relation that is closed under $\mathcal{F}$-contexts and under $\mathcal{T}(\mathcal{F}, X)$-substitutions.

Definition 3.2.5. Let $\mathcal{R}$ be a TRS. A term t rewrites to a term s, written as $t \xrightarrow{\pi}_{\mathcal{R}} s$ if there is a rule $l \to r \in \mathcal{R}$, a position $\pi \in \mathtt{Pos}(t)$ and a substitution $\sigma \in \mathtt{Subst}(\mathcal{T}(\mathcal{F}, X))$ such that $t|_\pi = \sigma(l)$ and $s = t[\sigma(r)]_\pi$. $t \to_{\mathcal{R}} s$ is called one-step rewrite relation. If the position at which the rewrite step is applied is of interest, we may write $t \xrightarrow{\pi} s$. If it is clear, which TRS is used, $\mathcal{R}$ can be omitted. If we want to make the rule used at some rewrite step explicit, we write $t \to_\alpha s$ where $\alpha = l \to r$. The rewrite relation induced by $\mathcal{R}$ is $\to^*_{\mathcal{R}}$, the transitive and reflexive closure of $\to_{\mathcal{R}}$.

Termination

One very important property of TRSs, respectively the rewrite relations induced by TRSs, is termination.

Definition 3.2.6 (Termination, SN)**.** A term $t \in \mathcal{T}(\mathcal{F}, X)$ is called *terminating with respect to* $\mathcal{R}$ if and only if there is no infinite rewrite sequence starting at t (i.e. there is no infinite sequence $t = t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \ldots$). A TRS $\mathcal{R}$ is called *terminating* (or *strongly normalizing*, *SN*) if all terms in $\mathcal{T}(\mathcal{F}, X)$ are terminating with respect to $\mathcal{R}$. A TRS that is not terminating is called *non-terminating*.

Termination of a TRS is closely related to its induced rewrite relation.

Definition 3.2.7 (Well-Foundedness)**.** A relation $\to$ is called *well-founded* if and only if there are no infinite descending chains.

A TRS $\mathcal{R}$ is terminating if and only if $\to^+_{\mathcal{R}}$ is well-founded.

Over the years since term rewriting was invented, many methods of proving termination of term rewriting systems have been introduced. One very successful approach is to use so-called reduction orders.

Definition 3.2.8 (Reduction Order)**.** A reduction order is a well founded order that is closed under substitutions and under contexts.

The following result (taken from [BN98]) describes how reduction orders can be used to prove termination of term rewriting systems:

Theorem 3.2.9 (Termination with Reduction Orders, [BN98])**.** *A term rewriting system $\mathcal{R}$ terminates iff there exists a reduction order $>$ that satisfies $l > r$ for all $l \to r \in \mathcal{R}$.*

The Dependency Pair Framework

One weakness of reduction orders is, that in practice most commonly used rewrite orders (e.g. RPO [Der82], LPO [KL80], and KBO [KB70]) are simplification orders ([Der82]), that can only be used to show termination of a limited subclass of terminating TRSs (simply terminating TRSs, see [BN98] for details). A different, more flexible, method of showing termination of term rewriting systems is the so-called *dependency pair framework*. The dependency pair (DP) framework was introduced by Giesl, Thiemann and Schneider-Kamp in [GTSK05]. It is based on the dependency pair approach by Arts and Giesl [AG00] and offers a modular method for proofing termination of term rewriting systems.

Definition 3.2.10 (Dependency Pairs, [GAO02])**.** Given a TRS $\mathcal{R}$ over a signature $\mathcal{F}$, the *set of dependency pairs* $DP(\mathcal{R})$ over the signature $\mathcal{F}^\# = \mathcal{F} \cup \left\{ f^\# \mid f \in \mathcal{F} \right\}$ is defined as follows:

$$DP(\mathcal{R}) = \left\{ l^\# \to u^\# \mid l \to r \in \mathcal{R}, u \trianglelefteq r, \mathtt{root}(u) \text{ defined}, u \not\trianglelefteq l \right\}$$

The original dependency pair framework from [GTSK05] is defined in a very general and flexible way. Here, we use a simplified version of the DP framework, similar to the one defined in [SM08], which suffices for our applications.

Definition 3.2.11 (DP problem, [SM08])**.** A pair of TRSs $(\mathcal{P}, \mathcal{R})$ is called a *DP problem*, if all the root symbols of rules in $\mathcal{P}$ occur *only* as root symbols in $\mathcal{P}$ and nowhere else in $\mathcal{P}$ or $\mathcal{R}$. The set $\mathcal{P}$ is called the set of dependency pairs of $(\mathcal{P}, \mathcal{R})$ and the set $\mathcal{R}$ is called the set of rules of $(\mathcal{P}, \mathcal{R})$.

Definition 3.2.12 (Finiteness of DP Problems)**.** A DP problem $(\mathcal{P}, \mathcal{R})$ is called *finite*, if there is no infinite rewrite sequence of the following shape:

$$s_1 \xrightarrow{\epsilon}_{\mathcal{P}} t_1 \to^*_{\mathcal{R}} s_2 \xrightarrow{\epsilon}_{\mathcal{P}} \to^*_{\mathcal{R}} t_2 \dots$$

where each t_i is terminating with respect to $\mathcal{R}$ (such a sequence is called a *minimal infinite rewrite sequence*).

As a direct consequence from the results in [AG00] we have the following theorem:

Theorem 3.2.13 ([AG00])**.** *A TRS $\mathcal{R}$ is terminating iff the DP problem $(DP(\mathcal{R}), \mathcal{R})$ is finite.*

The use of so-called DP processors makes the DP framework very flexible and modular. A DP processor `Proc` is a mapping from DP problems to sets of DP problems and a special symbol `no`, indicating non-finiteness. DP processors can be sound and complete.

Definition 3.2.14 (Soundness of DP Processors)**.** A DP processor `Proc` is called *sound* if, whenever $\mathtt{Proc}((\mathcal{P}, \mathcal{R})) \neq \mathtt{no}$ and all DP problems in $\mathtt{Proc}((\mathcal{P}, \mathcal{R}))$ are finite, then also $(\mathcal{P}, \mathcal{R})$ is finite.

Definition 3.2.15 (Completeness of DP Processors)**.** A DP processor `Proc` is called *complete* if, whenever $\mathtt{Proc}((\mathcal{P}, \mathcal{R})) = \mathtt{no}$ or at least one DP problem in $\mathtt{Proc}((\mathcal{P}, \mathcal{R}))$ is not finite, then also $(\mathcal{P}, \mathcal{R})$ is not finite.

The property of soundness is required for every DP processor in order to work as excepted in the DP framework. The property of completeness is required, if a DP processor should be able to prove non-finiteness of DP problems.

3.3 Context-Sensitive Rewriting

Context sensitive rewriting was introduced by Lucas in [Luc98]. It is a generalization of the term rewriting formalism, where in addition to a set of rules, one has to specify a *replacement map*. A replacement map is a function that describes at which positions in terms rewrite steps are allowed to be applied.

Definition 3.3.1 (Replacement Map, [Luc98])**.** Let $\mathcal{F}$ be a signature. A mapping $\mu\colon \mathcal{F} \mapsto 2^{\mathbb{N}}$ is a *replacement map* for the signature $\mathcal{F}$ if for all $f \in \mathcal{F}$: $\mu(f) \subseteq \{1, \ldots, \mathtt{ar}(f)\}$

Definition 3.3.2 (Replacement Condition, [Luc98])**.** Let $\mathcal{F}$ be a signature and μ be a replacement map. Let $t \in \mathcal{T}(\mathcal{F}, X)$ be a term. The *replacement condition* is a predicate $\gamma_{\mu,t}$ defined on the set of positions $\mathtt{Pos}(t)$ as follows:

$$\gamma_{\mu,t}(\epsilon)$$

$$\gamma_{\mu,f(t_1,\ldots,t_n)}(i\pi) \Leftrightarrow (i \in \mu(f)) \wedge \gamma_{\mu,t_i}(\pi)$$

Definition 3.3.3 (μ-Replacing Positions, [Luc98])**.** The set of all *μ-replacing positions* in a term t is defined as $\mathtt{Pos}^{\mu}(t) = \{\pi \mid \pi \in \mathtt{Pos}(t), \gamma_{\mu,t}(\pi)\}$. Analogously, $\mathtt{FPos}^{\mu}(t)$ denotes the set of all *μ-replacing non-variable positions* of t and $\mathtt{VPos}^{\mu}(t)$ denotes the set of all *μ-replacing variable positions* of t. Complementary to $\mathtt{Pos}^{\mu}(t)$, $\mathtt{FPos}^{\mu}(t)$ and $\mathtt{VPos}^{\mu}(t)$, the sets $\mathtt{Pos}^{\bar{\mu}}(t)$, $\mathtt{FPos}^{\bar{\mu}}(t)$ and $\mathtt{VPos}^{\bar{\mu}}(t)$ contain the *non-μ-replacing (non-variable/variable)-positions* of t.

Definition 3.3.4 (Variables and Symbols at (Non-)μ-replacing positions)**.** Let $t \in \mathcal{T}(\mathcal{F}, X)$ be a term and μ a replacement map. We define the following sets:

- $\mathtt{Vars}^{\mu}(t) = \{x \mid \pi \in \mathtt{VPos}^{\mu}(t), t|_{\pi} = x\}$
- $\mathtt{Vars}^{\bar{\mu}}(t) = \{x \mid \pi \in \mathtt{VPos}^{\bar{\mu}}(t), t|_{\pi} = x\}$
- $\mathtt{Func}^{\mu}(t) = \{f \mid \pi \in \mathtt{FPos}^{\mu}(t), \mathtt{root}(t|_{\pi}) = f\}$
- $\mathtt{Func}^{\bar{\mu}}(t) = \{f \mid \pi \in \mathtt{FPos}^{\bar{\mu}}(t), \mathtt{root}(t|_{\pi}) = f\}$

Definition 3.3.5 (CS-TRS)**.** A context-sensitive term rewriting system (CS-TRS) is a pair $(\mathcal{R}, \mu)$ where $\mathcal{R}$ is a TRS and μ is a replacement map for the signature of $\mathcal{R}$.

Definition 3.3.6 (Context-Sensitive Rewrite Relation [Luc98])**.** Let $(\mathcal{R}, \mu)$ be a CS-TRS. A term t μ-rewrites to a term s, written $t \hookrightarrow_{\mathcal{R},\mu} s$, if there is some position π such that $t \xrightarrow{\pi} s$ holds, and $\pi \in \mathtt{Pos}^{\mu}(t)$ (i.e. π is a μ-replacing position in t). The one-step context-sensitive rewrite relation of $\mathcal{R}$ wrt. μ is $\hookrightarrow_{\mathcal{R},\mu}$. The context-sensitive rewrite relation of $\mathcal{R}$ wrt. μ is $\hookrightarrow^*_{\mathcal{R},\mu}$. If $\mathcal{R}$ or μ are obvious from the context, they can be omitted. If the position at which the rewrite step is applied does not matter, it can also be omitted.

Termination of CS-TRSs

As in the case of TRSs, one can ask whether CS-TRSs are terminating. As before, termination is defined by means of the absence of infinite rewrite sequences in the rewrite relation induced by the CS-TRS. Since the rewrite relation of a CS-TRS is a restriction of the rewrite relation of the corresponding TRS, termination of CS-TRSs should follow more easily.

Definition 3.3.7 (Termination of CS-TRSs)**.** Let $(\mathcal{R}, \mu)$ be a CS-TRS and $t \in \mathcal{T}(\mathcal{F}, X)$ a term. t is called terminating with respect to $(\mathcal{R}, \mu)$ if there is no infinite rewrite sequence $t = t_1 \hookrightarrow_{\mathcal{R},\mu} t_2 \hookrightarrow_{\mathcal{R},\mu} \ldots$. A CS-TRS $(\mathcal{R}, \mu)$ is called terminating, if every term $s \in \mathcal{T}(\mathcal{F}, X)$ is terminating with respect to $(\mathcal{R}, \mu)$.

The Context-Sensitive Dependency Pair Framework

There are different approaches on how to generalize the DP framework to context sensitive rewriting. VMTL uses the improved version [AEF$^+$08] of the context-sensitive dependency pairs [AGL06] approach by Alarcon, Gutierrez and Lucas. The main difference between the original approach and the refined approach is, that the former uses collapsing dependency pairs, while the latter avoids this. Collapsing dependency pairs are problematic when generalizing existing ordinary (non-CS) DP processors to the context-sensitive case. In the refined version, many existing DP processors can be generalized to the context-sensitive case in an almost straightforward way [AEF$^+$08].

Definition 3.3.8 (μ-replacing (non-μ-replacing) subterm relation, [AGL06])**.** Let $(\mathcal{R}, \mu)$ be a CS-TRS. Let $s, t \in \mathcal{T}(\mathcal{F}, X)$ be terms. $\trianglelefteq_\mu$ and $\trianglelefteq_{\bar{\mu}}$ are binary relations on terms defined as follows:

$$\begin{array}{lcl} t \trianglelefteq_\mu s & \Leftrightarrow & s|_\pi = t \text{ for some } \pi \in \mathtt{Pos}^\mu(s) \\ t \trianglelefteq_{\bar{\mu}} s & \Leftrightarrow & s|_\pi = t \text{ for some } \pi \in \mathtt{Pos}^{\bar{\mu}}(s) \end{array}$$

In [AGL06], two distinct sets of dependency pairs for CS-TRSs $(\mathcal{R}, \mu)$ are defined: The set $DP_o((\mathcal{R}, \mu))$ of ordinary dependency pairs and the set $DP_c((\mathcal{R}, \mu))$ of collapsing dependency pairs.

Definition 3.3.9 ($DP_o((\mathcal{R}, \mu))$, $DP_c((\mathcal{R}, \mu))$, [AGL06])**.** Let $(\mathcal{R}, \mu)$ be a CS-TRS. The sets $DP_o((\mathcal{R}, \mu))$ and $DP_c((\mathcal{R}, \mu))$ are defined as follows:

$$\begin{array}{lcl} DP_o((\mathcal{R}, \mu)) & = & \left\{ l^\# \to u^\# \mid l \to r \in \mathcal{R}, u \trianglelefteq_\mu r, \mathtt{root}(u) \text{ defined} \right\} \\ DP_c((\mathcal{R}, \mu)) & = & \left\{ l^\# \to x \mid l \to r \in \mathcal{R}, x \trianglelefteq_\mu r, x \not\trianglelefteq_\mu l, x \in X \right\} \end{array}$$

In [AEF$^+$08] a method is introduced, that removes the collapsing dependency pairs from the set of dependency pairs of a CS-TRS, adding a new set $DP_u((\mathcal{R}, \mu))$ of non-collapsing dependency pairs (called unhiding dependency pairs) instead. This set is constructed from the set of collapsing dependency pairs. The dependency pairs in this set, together with the dependency pairs in $DP_o((\mathcal{R}, \mu))$ will form the set $DP((\mathcal{R}, \mu))$ of dependency pairs of a CS-TRS $(\mathcal{R}, \mu)$. First, we need the notion of hidden terms and hiding contexts:

Definition 3.3.10 (Hidden Term, [AEF$^+$08])**.** Let $(\mathcal{R}, \mu)$ be a CS-TRS. A term t is a *hidden term* if $\mathtt{root}(t)$ is a defined symbol in $\mathcal{R}$ and if there exists a rule $l \to r \in \mathcal{R}$ with $t \trianglelefteq_\mu r$.

Definition 3.3.11 (Hiding Context, [AEF$^+$08])**.** Let $(\mathcal{R}, \mu)$ be a CS-TRS. A function symbol f *hides* position i if there is a rule $l \to r \in \mathcal{R}$ with $f(r_1, \ldots, r_i, \ldots, r_n) \trianglelefteq_\mu r$, $i \in \mu(f)$, and r_i contains a defined symbol or a variable at a μ-replacing position. A context C is *hiding* iff $C = \Box$ or C has the form $f(t_1, \ldots, t_{i-1}, C', t_{i+1}, \ldots, t_n)$ where f hides position i and C' is a hiding context.

Definition 3.3.12 (Improved Context-Sensitive Dependency Pairs, [AEF$^+$08])**.** Let $(\mathcal{R}, \mu)$ be a CS-TRS. If $DP_c((\mathcal{R}, \mu))$ is not the empty set, a fresh unhiding tuple symbol U is introduced and the set $DP_u((\mathcal{R}, \mu))$ contains:

- $s \to U(x)$ for every $s \to x \in DP_c((\mathcal{R}, \mu))$
- $U(f(x_1, \ldots, x_i, \ldots, x_n)) \to U(x_i)$ for every $f \in \mathcal{F}^{(n)}$ and $1 \leq i \leq n$ where f hides position i
- $U(t) \to t^\#$ for every hidden term t

Otherwise, if $DP_c((\mathcal{R}, \mu)) = \emptyset$, $DP_u((\mathcal{R}, \mu))$ is empty.

The definition of CS-DP problems and CS-DP processors in [AEF$^+$08] is more general than the definition we use here (we do not use CS-DP processors for proving innermost termination).

Definition 3.3.13 (CS-DP Problem, CS-DP Processor)**.** A *CS-DP problem* is a tuple $(\mathcal{P}, \mathcal{R}, \mu)$, where $\mathcal{P}$ and $\mathcal{R}$ are TRSs and μ is a replacement map. A *CS-DP processor* is a function $\mathtt{Proc}$ from CS-DP problems to sets of CS-DP problems and a special symbol $\mathtt{no}$ indicating non-finiteness.

Definition 3.3.14 (Soundness of CS-DP Processors)**.** A CS-DP processor $\mathtt{Proc}$ is called *sound* if, whenever $\mathtt{Proc}((\mathcal{P}, \mathcal{R}, \mu)) \neq \mathtt{no}$ and all DP problems in $\mathtt{Proc}((\mathcal{P}, \mathcal{R}, \mu))$ are finite, then also $(\mathcal{P}, \mathcal{R}, \mu)$ is finite.

Definition 3.3.15 (Completeness of CS-DP Processors)**.** A CS-DP processor $\mathtt{Proc}$ is called *complete* if, whenever $\mathtt{Proc}((\mathcal{P}, \mathcal{R}, \mu)) = \mathtt{no}$ or at least one DP problem in $\mathtt{Proc}((\mathcal{P}, \mathcal{R}, \mu))$ is not finite, then also $(\mathcal{P}, \mathcal{R}, \mu)$ is not finite.

In [AEF$^+$08], termination of a CS-TRS is characterized by the absence of infinite chains. For consistency with the definition of the non-context-sensitive DP framework in the previous section and for some proofs in later chapters, we also give a different characterization and argue that it is equivalent.

Definition 3.3.16 ($(\mathcal{P}, \mathcal{R}, \mu)$-Chain, Finiteness of a CS-DP problem, [AEF$^+$08])**.** Let $(\mathcal{P}, \mathcal{R}, \mu)$ be a CS-DP problem. A $(\mathcal{P}, \mathcal{R}, \mu)$-*chain* is a sequence of dependency pairs $s_1 \to t_1, s_2 \to t_2, \cdots \in \mathcal{P}$ (pairwise variable disjoint), where there exists a substitution σ, such that for all i, $\sigma(t_i) \hookrightarrow^*_{\mathcal{R},\mu} \sigma(s_{i+1})$ holds. A $(\mathcal{P}, \mathcal{R}, \mu)$-chain is called *minimal* if for all i, the term t_i terminates

wrt. $(\mathcal{R}, \mu)$. A CS-DP problem $(\mathcal{P}, \mathcal{R}, \mu)$ is called finite if no infinite minimal $(\mathcal{P}, \mathcal{R}, \mu)$-chain exists.

For some proofs, a different (equivalent) characterization of finiteness of a CS-DP problem is convenient.

Definition 3.3.17 (Finiteness of a CS-DP problem)**.** A CS-DP problem $(\mathcal{P}, \mathcal{R}, \mu)$ is finite iff there is no infinite rewrite sequence of the following shape:

$$s_1 \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} t_1 \hookrightarrow^*_{\mathcal{R},\mu} s_2 \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} t_2 \hookrightarrow^*_{\mathcal{R},\mu} \dots$$

where each t_i is terminating with respect to $(\mathcal{R}, \mu)$ (such a sequence is called a *minimal infinite rewrite sequence*).

Proposition 3.3.18. *The two characterizations of finiteness of a CS-DP problem $(\mathcal{P}, \mathcal{R}, \mu)$ are equivalent. That is, $(\mathcal{P}, \mathcal{R}, \mu)$ is finite wrt. Definition 3.3.16 if and only if it is finite wrt. Definition 3.3.17.*

Proof. Suppose, a CS-DP problem $(\mathcal{P}, \mathcal{R}, \mu)$ is not finite according to Definition 3.3.17. Then there is an infinite rewrite sequence of the following shape:

$$s_1 \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} t_1 \hookrightarrow^*_{\mathcal{R},\mu} s_2 \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} t_2 \hookrightarrow^*_{\mathcal{R},\mu} \dots$$

where each t_i terminates wrt. $(\mathcal{R}, \mu)$. Therefore, we have $s_i \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} t_i$ for all $i \geq 1$, which implies, that for each of these steps, there has to be a rule $l_i \to r_i \in \mathcal{P}$ and substitution σ_i such that $s_i = \sigma_i(l_i)$ and $t_i = \sigma_i(r_i)$ (because, by assumption the rule is applied at root position). Without loss of generality, we can assume that the rules $l_i \to r_i$ do not share any variables and the domains of the substitutions σ_i are pairwise disjoint. Therefore, we can define a new substitution σ that combines all σ_i. From $t_i \hookrightarrow^*_{\mathcal{R},\mu} s_{i+1}$ we know $\sigma(l_i) \hookrightarrow^*_{\mathcal{R},\mu} \sigma(r_{i+1})$. Furthermore, all t_i are terminating wrt. $(\mathcal{R}, \mu)$, so also all $\sigma(r_i)$ are terminating wrt. $(\mathcal{R}, \mu)$. Thus, we have an infinite minimal $(\mathcal{P}, \mathcal{R}, \mu)$-chain $l_1 \to r_1, l_2 \to r_2, \dots$.
Now, suppose $(\mathcal{P}, \mathcal{R}, \mu)$ is not finite according to Definition 3.3.16. We know, that there is a sequence of rules from $\mathcal{P}$: $s_1 \to t_1, s_2 \to t_2, \dots$ and a substitution σ such that for all $i \geq 1$, $\sigma(t_i) \hookrightarrow^*_{\mathcal{R},\mu} \sigma(s_{i+1})$ and $\sigma(t_i)$ terminates wrt. $(\mathcal{R}, \mu)$. Since CS-rewriting is closed under substitutions([Luc95]), it follows that $\sigma(s_i) \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} \sigma(t_i)$ for all $i \geq 1$. Thus, we obtain the following infinite rewrite sequence:

$$\sigma(s_1) \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} \sigma(t_1) \hookrightarrow^*_{\mathcal{R},\mu} \sigma(s_2) \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} \sigma(t_2) \hookrightarrow^*_{\mathcal{R},\mu} \dots$$

where each $\sigma(t_i)$ $(i \geq 1)$ terminates wrt. $(\mathcal{R}, \mu)$. □

Theorem 3.3.19 ([AEF$^+$08])**.** *A CS-TRS $(\mathcal{R}, \mu)$ is terminating if and only if the CS-DP problem $(DP((\mathcal{R}, \mu)), \mathcal{R}, \mu')$ is finite, where $\mu'(f) = \mu(f)$ and $\mu'(f^\#) = \mu(f)$ for all $f \in \mathcal{F}$ and, if $DP_c(\mathcal{R}, \mu) \neq \emptyset$ then $\mu'(U) = \emptyset$.*

3.4 Conditional Rewriting

In conditional rewriting, applicability of rules depends on additional conditions that are formulated as part of the rules. For a comprehensive introduction to conditional rewriting we refer to [Ohl02]. Here we just state the most important definitions needed for working with VMTL. A conditional term rewriting system (CTRS) consists of rules of the following shape:

$$l \to r \Leftarrow s_1 \to^* t_1, \ldots, s_n \to^* t_n$$

The part after $\Leftarrow$ in the rule are the conditions. The conditions may also be empty, in which case $\Leftarrow$ can be omitted. Such a system is called *oriented* CTRS (because $\to^*$ is used in the conditions). For use with VMTL we are interested in so-called oriented 3-CTRSs. A 3-CTRS is a CTRS where for each rule the following restriction concerning the occurrence of variables holds: $\mathtt{Vars}(l) \cup \bigcup_{i=1}^{n} (\mathtt{Vars}(s_i) \cup \mathtt{Vars}(t_i)) \subseteq \mathtt{Vars}(r)$. That is, in a 3-CTRS, variables in the right-hand-side of rules must appear either in the left-hand-side or in the conditions. A deterministic CTRS (DCTRS) is an oriented 3-CTRS, where for each rule, the following additional condition holds: $\mathtt{Vars}(s_i) \subseteq \mathtt{Vars}(l) \cup \bigcup_{j=1}^{i-1} \mathtt{Vars}(t_j)$.

In the following, when talking about CTRSs, we always mean oriented 3-CTRSs.

Definition 3.4.1 (Conditional Rewrite Relation)**.** Let $\mathcal{R}$ be a oriented CTRS. The rewrite relation $\to_{\mathcal{R}}$ is defined inductively. For the base case we consider the TRS $\mathcal{R}_0 = \emptyset$. In the induction step, we define the following TRSs:

$$\mathcal{R}_k = \left\{\sigma(l) \to \sigma(r) \mid l \to r \Leftarrow s_1 \to^* t_1, \ldots, s_n \to^* t_n \in \mathcal{R}, \sigma(s_i) \to_{\mathcal{R}_{k-1}} \sigma(t_i), 1 \leq i \leq n\right\}$$

Finally, we define $\to_{\mathcal{R}} = \bigcup_{i \geq 0} \to_{\mathcal{R}_i}$.

The inductive definition of the rewrite relation leads to a subtle problem regarding termination. Consider the following CTRS $\mathcal{R}$ from [SG10, Sch11]:

$$a \to b \Leftarrow a \to^* b$$

In this example, the rewrite relation is empty, thus $\mathcal{R}$ is terminating when considering the ordinary notion of termination. However, tools that try to apply the rule might loop when trying to figure out if the condition for the rule is satisfied. Therefore, when dealing with CTRSs, one usually needs a stronger notion of termination that also captures these cases. To this end, in [LMM05] the notion of *operational termination* is introduced which is what we are interested in when proving termination of CTRSs.

DCTRSs can be transformed into context-sensitive TRSs by the so-called *context-sensitive unravelling* transformation [SG10, Sch11]. In [SG10, Sch11] it is shown, that if the CS-TRS obtained by this transformation is terminating, then the original DCTRS is context-sensitive quasi-reductive which in turn implies that it is operationally terminating.

Definition 3.4.2 (Context-Sensitive Unravelling of DCTRSs, [SG10, Sch11])**.** Let $\mathcal{R}$ be a DCTRS over a signature $\mathcal{F}$. The context-sensitive unravelling of $\mathcal{R}$ is the CS-TRS

$(\mathrm{U}_{\mathrm{CS}}(\mathcal{R}), \mu_{U(\mathcal{F})})$. For every conditional rule $\alpha\colon l \to r \Leftarrow s_1 \to^* t_1, \ldots, s_n \to^* t_n$, n new function symbols $U_i^\alpha (1 \leq i \leq n)$ are introduced. For all conditional rules α, $\mathrm{U}_{\mathrm{CS}}(\mathcal{R})$ contains the following unconditional rules:

$$\begin{aligned} l &\to U_1^\alpha(s_1, \mathcal{V}ar(l)) \\ U_1^\alpha(t_1, \mathcal{V}ar(l)) &\to U_2^\alpha(s_2, \mathcal{V}ar(l), \mathcal{EV}ar(t_1)) \\ &\vdots \\ U_n^\alpha(t_n, \mathcal{V}ar(l), \mathcal{EV}ar(t_1), \ldots, \mathcal{EV}ar(t_{n-1})) &\to r \end{aligned}$$

where $\mathcal{V}ar(t)$ is an arbitrary but fixed sequence of the variables occurring in t, and $\mathcal{EV}ar(t_i)$ is an arbitrary but fixed sequence of variables occurring in the set $\mathtt{Vars}(t_i) \setminus (\mathtt{Vars}(l) \cup \bigcup_{j=1}^{i-1} \mathtt{Vars}(t_j))$. Unconditional rules are inherited from $\mathcal{R}$ into $\mathrm{U}_{\mathrm{CS}}(\mathcal{R})$ without changes. $\mathrm{U}_{\mathrm{CS}}(\mathcal{R})$ is a TRS over the signature $U(\mathcal{F})$ which extends $\mathcal{F}$ by all newly introduced symbols. The replacement map $\mu_{U(\mathcal{F})}$ is defined as follows: $\mu_{U(\mathcal{F})}(f) = \{1 \ldots n\}$ for all $f \in \mathcal{F}$ with $\mathtt{ar}(f) = n$ and $\mu_{U(\mathcal{F})}(u) = \{1\}$ for all newly introduced symbols u.

Example 3.4.3. Applying the transformation on the example from before yields the following CS-TRS:

$$\begin{aligned} a &\to U_1^1(a) \\ U_1^1(b) &\to b \end{aligned}$$

with the replacement map μ where $\mu(U_1^1) = \{1\}$. This CS-TRS is obviously non-terminating, since the lhs of the first rule is embedded in its rhs at a μ-replacing position. Thus, by the results from [SG10, Sch11] we can conclude, that the original CTRS from the example is not operationally terminating.

CHAPTER 4

VMTL

In this chapter we discuss some of the features of VMTL. In particular, we will see how proof strategies are executed by VMTL. After that, we present the extensions that are added to VMTL as part of this thesis. The most important extensions are the support for direct proof methods and a new strategy language that supports these new methods. In the last section we give a short overview of VMTL from a programmer's point of view, and discuss how the program logic of VMTL is organized. This section may be helpful when planning to extend VMTL further.

4.1 VMTL

VMTL (Vienna Modular Termination Laboratory [SG09]) is a tool for automatic termination proofs of term rewriting systems, context-sensitive term rewriting systems and conditional term rewriting systems. As the name suggests, one big emphasis of VMTL is modularity. New proof techniques and transformations can be added without the need of recompiling or changing the core VMTL system. The core system also provides functions implementing the most commonly used operations on terms, TRSs, etc. which can be used when implementing new proof techniques.

The language of implementation is Java (Version 1.5). Currently (considering all modifications and extensions made as part of this thesis) the core VMTL system consists of 61 classes and about 30.500 lines of code. The currently implemented set of proof techniques contribute another 29 classes and about 14.700 lines of code and the VMTL Sat Solving Facility (see Chapter 5) adds another 42 classes and about 12.700 lines of code. Since Java is an object-oriented language, modularity (i.e. adding new proof techniques) is realized in a very natural way. When a new proof-technique or transformation scheme is added, a new class (implementing the corresponding Java interface) is created, and the resulting ".class"-file is loaded by VMTL at runtime whenever it is needed.

Strategies for the DP Analysis

Proof search in VMTL is done in the DP framework (cf. Chapter 3.1). Such a proof search will from now on be called *DP analysis*. All currently implemented proof techniques are (CS-)DP processors. Due to the flexibility of the DP framework regarding the order of application of DP processors, it is very important to choose a reasonable order in which the processors are applied. For example there are DP processors that split a given DP problem into a set of smaller DP problems. DP processors of this kind will usually be used first. Then the resulting smaller DP problems will be treated by different DP processors, until termination is proved. Furthermore, in VMTL the behaviour of DP processors can be controlled by additional parameters. For example, the user can choose, whether the RPOS reduction pair processor should use usable rules or not.

VMTL allows the user to control the order in which the different DP processors are applied, as well as the parameters for each application of a DP processor, by a so-called strategy which has to be provided by the user (this is the reason that, strictly speaking, VMTL is a *semi*-automatic termination prover). The choice of strategy should depend on the TRS which should be proven (or disproven) terminating. However, sometimes one just wants a quick attempt at proving termination of some system, without bothering to think of a strategy first. For these cases, VMTL assumes a good "general-purpose" strategy that has proven useful in many practical tests and competitions.

For the biggest part, the strategies used by VMTL are quite intuitive. Nevertheless, there are some things that could be implemented in different ways. So, in order to have a rigid formal definition of how strategies in VMTL are executed, we specify how strategies are handled by VMTL. Abstractly, a strategy can be considered as a tree with ordered branches (called a strategy tree), where each node can be either a DP processor node or a group node. DP processor nodes are always leaves in the tree, while group nodes can have an arbitrary number of child nodes (at least one is required, however). We formally define how strategy trees are built and how they are "executed" by VMTL.

Strategy Trees

Here, we discuss how well-formed strategy trees are built. Note, that the description of strategy trees given here is a high-level description, based on the notion of mathematical trees. This concept will be used when describing the proof search in VMTL and for formal reasoning. The actual syntax of the *strategy file*, defining a strategy tree, that is given to VMTL will be based on XML and will be described later in this chapter.

In our setting, a node can have an arbitrary number of parameters (key-value mappings) and an arbitrary number of child nodes. Furthermore, each node has a specific type that defines how VMTL should handle that node. We distinguish two special types of nodes that can occur in strategy trees: DP processor nodes and DP group nodes. Both DP processor nodes and DP group nodes have at least two parameters: The parameter `time` and the parameter `runs`, both with an integer value. These two parameters specify details about the proof search. Their meaning will be discussed shortly in the next subsection dealing with the execution of strategy trees.

In the following, let Str be the set of all strings over the alphabet of all non-whitespace

character symbols.

Definition 4.1.1 (Parameter)**.** A parameter is a triple (k, D, def), where k (the key) is a textual string from Str, D is the domain of elements that the value of the parameter may assume and def is a default value (an element from D). We define the two standard parameters, that are used in every node as follows:

- **time** $= (\texttt{time}, \mathbb{N}, 0)$ for specifying the maximum time VMTL should work on executing the node.
- **runs** $= (\texttt{runs}, \mathbb{N}, 1)$ for specifying how often the node should be executed by VMTL.

Definition 4.1.2 (DP Strategy Node)**.** A DP strategy node is a tuple (t, P, PA, C) where t is a textual string from Str describing the type of the node, P is a set of parameters available for this node and PA is a parameter assignment (a set of pairs (k, v) where k is the key from some parameter in P and v is some element from the domain of that parameter, with all k pairwise distinct). Furthermore, C is a list of child nodes (i.e. DP strategy nodes).

Definition 4.1.3 (DP Processor Node)**.** A DP processor node is a DP strategy node $(t, P, PA, \emptyset)$ where t is the name of some DP processor (we assume, that no processor uses the empty string ϵ as its name and that different DP processors use different names). If the implementation of the referenced DP processor in VMTL depends on parameters $(k_1, D_1, def_1), \ldots, (k_n, D_n, def_n)$, then $P = \{\textbf{time}, \textbf{runs}\} \cup \{(k_i, D_i, def_i) \mid 1 \leq i \leq n\}$. C is empty for DP processor nodes (i.e. they are leaf nodes in strategy trees).

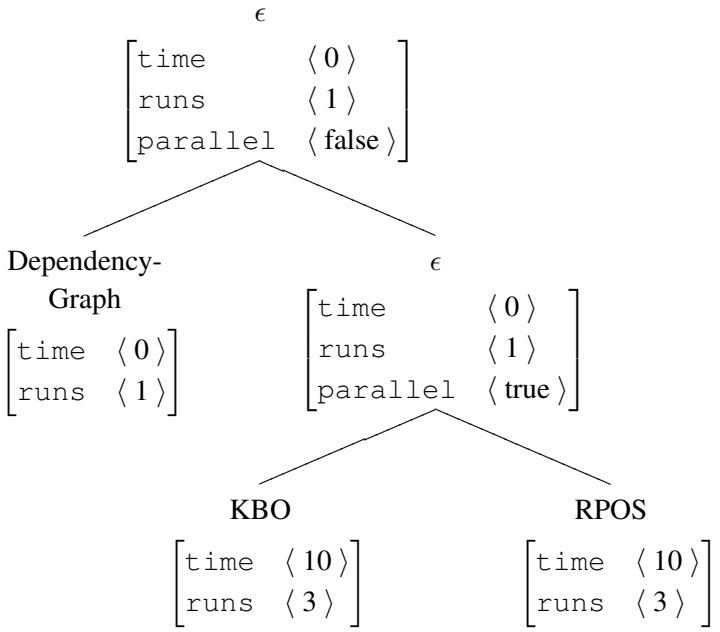
Definition 4.1.4 (DP Group Node)**.** Groups can be executed in two ways: sequentially or parallel. The way they are executed is defined by two additional parameters:

- **parallel** $= (\texttt{parallel}, \{\textbf{true}, \textbf{false}\}, \textbf{false})$ for specifying whether this is a sequential or a parallel group.
- **dp_evaluator** $= (\texttt{dp\_evaluator}, Str, \texttt{default\_evaluator})$ is needed only in parallel groups to determine the best result.

A DP group node is a DP strategy node $(\epsilon, \{\textbf{time}, \textbf{runs}, \textbf{parallel}, \textbf{dp\_evaluator}\}, PA, C)$ where the list C contains at least one child node.

Definition 4.1.5 (DP Analysis Strategy Tree)**.** A DP analysis strategy tree is a tree that is rooted by some DP strategy node.

Example 4.1.6.



This strategy tree represents the very simple strategy, where first the dependency graph processor is applied. After that the KBO and RPOS processor are tried in parallel. Each of these processors may run three times.

The next subsection describes how VMTL executes strategy trees.

Execution of Strategy Trees

In this subsection, we give a very formal definition, of how VMTL executes a given strategy, accompanied by an informal description, that is probably sufficient for using VMTL but might not be as precise.

When searching for a proof, VMTL "executes" the root node of a DP analysis strategy tree. Formally, the execution of a DP strategy node n amounts to applying a mapping ϕ_n from sets of DP problems to sets of DP problems. However, there is one problem when trying to formalize this mapping: Every strategy node provides two parameters, one of which is `time`. This parameter defines a timeout for the calculation of the corresponding node. If this parameter is zero, then there is no time-limit. However, if it is not zero (let it be t), VMTL abandons the calculation if after t seconds no result is found. To handle this source of non-determinism and formally model this behaviour, we define for each node n and set of DP problems I, the constant $t_{\phi_n(I)}$ that specifies the time that VMTL will require to find the result of $\phi_n(I)$ (ignoring time constraints). Of course, this does not work in practice. Instead, VMTL uses a Java-Timer that signals the calculation to stop after a certain time has passed.

Definition 4.1.7 (Parameter Assignment in a Strategy Node)**.** Let $n = (t, P, PA, C)$ be a strategy node. The parameter assignment function PA_n maps all keys k appearing as keys of parameters in P to their assigned value in the node n. For all k that are keys of some parameter in P define:

$$PA_n(k) = \begin{cases} v & \text{if there is some pair } (k, v) \in PA \\ def & \text{otherwise, if } (k, D, def) \in P \text{ for some set } D \end{cases}$$

Definition 4.1.8 (Strategy Node Execution Function)**.** Let $n = (t, P, PA, C)$ be a strategy node. The execution function ϕ_n maps sets of DP problems to sets of DP problems. The calculation of $\phi_n(I)$ for some input set I is described successively in the remainder of this subsection.

First, VMTL checks, whether the node n is a DP processor node or a DP group node. If the parameter `runs` is greater than one, the node has to be executed repeatedly. Thus we can define:

$$\phi_n(I) = \begin{cases} \overbrace{\phi_n^G(\phi_n^G(\dots(I)\dots))}^{r \text{ times}} & \text{if n is a DP group node and } PA_n(\texttt{runs}) = r \\ \overbrace{\phi_n^{DP}(\phi_n^{DP}(\dots(I)\dots))}^{r \text{ times}} & \text{if n is a DP processor node and } PA_n(\texttt{runs}) = r \end{cases}$$

In the case, that $n = (t, P, PA, \emptyset)$ is a DP processor node, VMTL executes the DP processor `Proc` that is referenced by the name in t for each problem in the set of input problems. If there is more than one DP problem in the input set, the time available at the current node is divided evenly, so every application of the DP processor gets the same amount of time. Formally, in this case we have $\phi_n^{DP}(\{(\mathcal{P}_i, \mathcal{R}_i) \mid 1 \le i \le k\}) = \phi_{n'}^{DP}(\{(\mathcal{P}_1, \mathcal{R}_1)\}) \cup \dots \cup \phi_{n'}^{DP}(\{(\mathcal{P}_k, \mathcal{R}_k)\})$ where $n' = (t, P, PA', \emptyset)$ and $PA' = PA$ except for the parameter assignment $(\texttt{time}, t)$ in PA which becomes $(\texttt{time}, \lceil t/k \rceil)$ in PA'. If the input-set contains only one DP problem, VMTL returns the input system in case a timeout occurs, and otherwise it returns the result of the application of the corresponding DP processor. Therefore, we define:

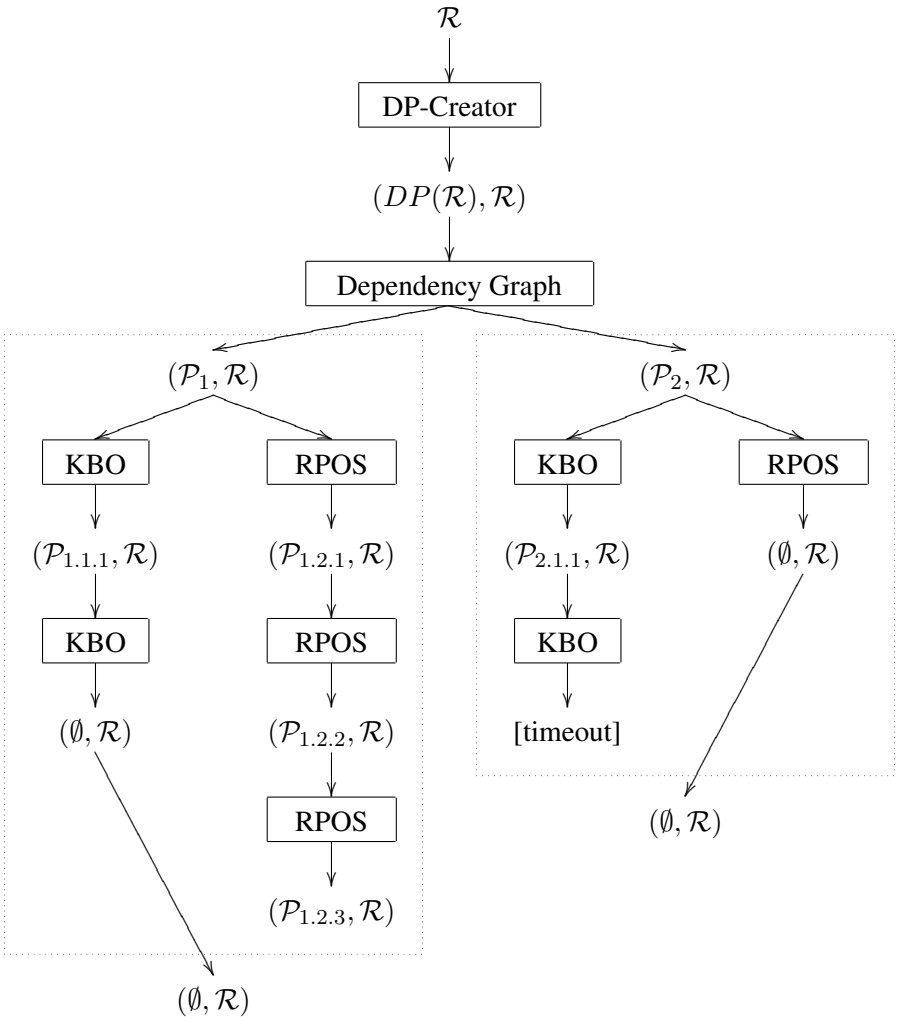
$$\phi_n^{DP}(\{(\mathcal{P}, \mathcal{R})\}) = \begin{cases} \{(\mathcal{P}, \mathcal{R})\} & \text{if } t_{\phi_n^{DP}(\{(\mathcal{P},\mathcal{R})\})} > PA_n(\texttt{time}) \\ \texttt{Proc}(\{(\mathcal{P}, \mathcal{R})\}) & \text{otherwise} \end{cases}$$

If a DP group node $n = (\epsilon, P, PA, C)$ is executed, VMTL checks, whether the group is a sequential group or a parallel group. If it is a sequential group (i.e. $PA_n(\texttt{parallel}) = \textbf{false}$), all child nodes have to be executed one after another, where the input of the first child node in the list is the input set I, the result of the first child node is used as the input of the second child node and so on. The sequential processing shares a common timeout. So it might happen, that the first few child nodes already consume all time that is available at this node. In this case, the remaining child nodes are not executed and the currently obtained result is returned. Let the list of child nodes be $C = [n_1, \dots, n_k]$ where $n_1 = (t_1, P_1, PA_1, C_1), \dots, n_k = (t_k, P_k, PA_k, C_k)$. For all i with $1 \le i \le k$ define $n_i' = (t_i, P_i, PA_i', C_i)$ with PA'_{n_i} = PA_{n_i} except that $(\texttt{time}, t_i)$ from PA_{n_i} becomes $(\texttt{time}, \min\{t_i - t_\Sigma, t - t_\Sigma\})$ in PA'_{n_i}, where $t_\Sigma = t_{\phi_{n_{i-1}}(\phi_{n_{i-2}}(\dots\phi_{n_1}(I))\dots)}$

and $t = PA_n(\texttt{time})$. Now choose the highest j such that for n'_j, $PA_{n'_j}(\texttt{time})$ is still greater than zero. Then $\phi_n(I) = \phi_{n_{j-1}}(\phi_{n_{j-2}}(\dots \phi_{n_1}(I) \dots))$.

Finally, if the executed node $n = (\epsilon, P, PA, C)$ is a parallel DP group node (i.e. if the parameter `parallel` is **true** in n), VMTL executes all child nodes in parallel with the same input. The result of the execution is then the most simple result that any of the child nodes yielded. Which one is the most simple is determined with the evaluation function referenced by the parameter `dp_evaluator` (The evaluation function returns a numerical value for any given set of DP problems such that a lower number means that this set of DP problems is potentially easier to prove terminating than a set of DP problems that gives a higher number). As before, let the list of child nodes be $C = [n_1, \dots, n_k]$ where $n_1 = (t_1, P_1, PA_1, C_1), \dots, n_k = (t_k, P_k, PA_k, C_k)$. Furthermore, let $eval$ be the evaluation function referenced by the property $PA_n(\texttt{dp\_evaluator})$. Then $\phi_n(I) = \phi_{n_i}(I)$ if $eval(\phi_{n_i}(I)) \leq eval(\phi_{n_j}(I))$ for all j with $1 \leq j \leq k$. If there are more possible choices for i, it is not defined which one is chosen.

Example 4.1.9.



This tree visualizes a possible execution of the strategy from Example 4.1.6 for some TRS $\mathcal{R}$. First, the TRS is transformed into the corresponding DP problem $(DP(\mathcal{R}), \mathcal{R})$ (This is done implicitly and is not part of the strategy). After that, the dependency graph processor is applied which yields two new DP problems $(\mathcal{P}_1, \mathcal{R})$ and $(\mathcal{P}_2, \mathcal{R})$. Both problems are examined in parallel by the KBO processor and the RPOS processor, which successively try to simplify the problem (each of these may run up to three times according to the strategy). In the end, for the first problem, $(\mathcal{P}_1, \mathcal{R})$, KBO finds a proof of termination, while RPOS did not succeed after three successive attempts. Thus the parallel group returns the more simple result, which will (depending on the evaluation function that is used used) most probably be $(\emptyset, \mathcal{R})$. For the second problem, RPOS succeeds after the first attempt and KBO times out. Again, the parallel

group returns the most simple output $(\emptyset, \mathcal{R})$. Thus for both branches, finiteness could be proven. Since all DP processors used in VMTL are sound, we can conclude that $\mathcal{R}$ is terminating.

Context-Sensitive Rewriting

VMTL supports termination proofs for context-sensitive TRSs. As in the context-free case, proof search is based on Dependency Pairs. For the context-sensitive version, the improved CS-DP framework from [AEF$^+$08] is used (See Chapter 3.3 for a quick summary). The improved CS-DP framework is closely related to the regular DP framework. Therefore, many existing DP processors can be adapted for the context-sensitive version very easily and, more importantly, most CS-DP processors are generalizations of their corresponding regular DP processors, so one gets those for free as a special case. However, there are some DP processors that require some more effort to adapt them to CS-DP processors. One example is the reduction pair processor with usable rules, where the usable rules cannot easily be generalized to the context-sensitive case. Most CS-DP processors in VMTL that utilize usable rules, incorporate the CS-usable rules described in [AEF$^+$08].

Having said this, all processors currently implemented in VMTL are really CS-DP processors that treat non context-sensitive DP problems as a special case (mostly by using the complete replacement-map (i.e. the replacement map, where every position is allowed), but in some cases, like usable-rules, by making a case distinction). This makes the use of VMTL very convenient, because strategies defined for termination proofs of context-sensitive TRSs can be used without change for termination proofs of context-free TRSs and vice versa (although, in most cases it is probably still better to use a strategy tailored to the TRS at hand).

Conditional Rewriting

For proving operational termination (cf. [LMM05]) of CTRSs, VMTL uses the context-sensitive unravelling transformation ([SG10]) in order to translate CTRSs into CS-TRSs. Afterwards, the CS-DP framework is used as described to try to prove termination of the obtained CS-TRS. As shown in [SG10], it is sufficient to show termination over the original signature of the CTRS (i.e. the symbols introduced by the transformation need not be considered).

TRS-to-TRS Transformations

The context-sensitive unravelling transformation is implemented as a TRS-to-TRS transformation. It is also possible to specify additional transformation operations on TRSs that can be applied on TRSs before the proof attempt is started. As in the case of unravelling, these transformations may also change the type of TRS (i.e. transform a CTRS into a CS-TRS, a CS-TRS into a TRS and so on).

Output Processing

VMTL provides an interface for specifying the output (i.e. proof details) in a uniform way. This interface is used by all DP processors. Output specified in this way can be processed by special

output processors in VMTL. For example, the default output processor implemented in VMTL generates HTML output. If needed, one could easily add an output processor that generates raw text output or even LaTeX output.

4.2 Extensions of VMTL

As part of this thesis, support for direct proof methods is added to VMTL. This section covers the integration of these methods and the corresponding extension of the strategy language. Another part of this thesis is the integration of two concrete, new (to VMTL) proof methods: KBO and root-labeling (as direct methods and as DP processors). The integration of these methods is covered in dedicated chapters (Chapter 7 for KBO, Chapter 6 for root-labeling).

To ease the specification and solving of SAT and SMT solving tasks for VMTL processors and methods, a framework for SAT/SMT solving is developed. This framework is the basis for the new KBO processor and can be used for any processors utilizing SAT or SMT solving, which are added in the future. A detailed description of the SAT/SMT solving framework can be found in Chapter 5.

Direct Methods

As mentioned before, the proof search in VMTL is based on the dependency pair framework (DP framework). One goal of this work is to add support for direct proof methods (direct methods) in VMTL which can be used as preprocessing before engaging the DP framework. In contrast to the DP processors used in the dependency pair framework, direct proof methods do not operate on DP problems, but directly examine a given rewrite system.

Definition 4.2.1. A mapping $\mathcal{DM}$ from rewrite systems to rewrite systems and a new symbol `no` (indicating non-termination) is called a direct method (abbreviated DM), if the following properties are satisfied for all rewrite systems $\mathcal{R}$ and $\mathcal{S}$:

- if $\mathcal{DM}(\mathcal{R}) = \texttt{no}$, then $\mathcal{R}$ is non-terminating.
- if $\mathcal{DM}(\mathcal{R}) = \mathcal{S}$, then if $\mathcal{S}$ is terminating also $\mathcal{R}$ is terminating.
  In particular, this means, that if $\mathcal{DM}(\mathcal{R}) = \emptyset$, then $\mathcal{R}$ is terminating.

By the definition of direct methods, every direct method is *sound*. A direct method is called *complete*, if the following additional condition holds:

- if $\mathcal{DM}(\mathcal{R}) = \mathcal{S}$, then if $\mathcal{S}$ is non-terminating, also $\mathcal{R}$ is non-terminating.

Given the input system $\mathcal{R}$, most implementations of direct methods will just return the empty set, if the system could be proven terminating, or the input system $\mathcal{R}$ itself, if no proof is found. However, the above definition also allows direct methods to return a special symbol `no`, if non-termination is detected. More advanced implementations can also return a different rewrite system $\mathcal{S}$ with the same termination-properties as $\mathcal{R}$. In most cases $\mathcal{S}$ will be a subset of $\mathcal{R}$. In

any case, proving termination of $\mathcal{S}$ should in some sense be easier than proving termination of $\mathcal{R}$.

Similar to the DP processors and CS-DP processors, direct methods can also be implemented as context-sensitive direct methods (CS-DMs) that operate on CS-TRSs instead of TRSs.

Since one big emphasis of VMTL is in modularity, direct methods are also added in a very modular way. This means, that it is easy to add new direct methods later on, without having to recompile VMTL every time. To this end, a set of Java-interfaces (extending the current VMTL-API) is provided, which can be implemented by new direct methods. New methods, implementing these interfaces are available at runtime for proof search.

Like the DP processors, every direct method can provide a number of parameters through its interface, which allow to configure the mechanics behind the proof method.

If a direct proof methods succeeds in proving termination (or non-termination) of a rewrite system or finds a simpler system with equal termination properties, it should provide appropriate output information. The output should contain all details that are necessary for the user to understand why the given system is indeed terminating (or non-terminating, resp.). The output can be specified using the same output specification classes that are used for DP processor implementations. Thus the output-processors can handle the output of both, direct methods and DP processors in a generic way.

If no proof is found (i.e. the output of the direct method is not the empty set or `no`), the output-system is passed on to the next method according to the defined strategy (the extended strategy is covered in Section 4.2 in detail). The output of the last direct method in the preprocessing strategy is then transformed into the corresponding DP problem, which is used as input for the DP analysis (according to the DP analysis strategy).

Extensions of the Strategy Specification

In order to support direct methods, a separate strategy for the direct proof attempt can be provided. This strategy will be executed before the DP analysis strategy is executed. The direct method proof attempt will thus be called "preprocessing" in the following. Subsequent to the preprocessing, the DP analysis strategy is executed.

Both strategies are optional: If only the DP analysis strategy is provided, no preprocessing is done and the proof attempt starts directly with the DP analysis step. On the other hand, if no DP analysis strategy is provided, the proof attempt stops as soon as preprocessing is finished. If no strategy at all is provided, then a default strategy (including preprocessing and DP analysis) is assumed.

The structure of the strategy specification for preprocessing is quite similar to that of the previously discussed DP analysis strategy in Section 4.1, but due to the simpler notion of direct methods (as compared to DP processors), the strategy also is a little simpler.

Strategy Trees for Preprocessing

The definition of preprocessing strategy trees is very similar to that of strategy trees for the DP analysis from Section 4.1. Let parameters be defined as in Definition 4.1.1. We adopt the notion of DP strategy nodes, DP processor nodes and DP group nodes for preprocessing as follows:

Definition 4.2.2 (DM Strategy Node)**.** A DM strategy node is a tuple (t, P, PA, C) where t is a textual string from Str describing the type of the node, P is a set of parameters available for this node and PA is a parameter assignment (a set of pairs (k, v) where k is the key from some parameter in P and v is some element from the domain of that parameter, all k pairwise distinct). Furthermore, C is a list of child nodes (i.e. DM strategy nodes).

Definition 4.2.3 (DM Node)**.** A DM node is a DM strategy node $(t, P, PA, \emptyset)$ where t is the name of some direct method (we assume, that no method uses the empty string ϵ as its name). If the implementation of the referenced direct method in VMTL depends on parameters $(k_1, D_1, def_1), \ldots, (k_n, D_n, def_n)$, then $P = \{\textbf{time}, \textbf{runs}\} \cup \{(k_i, D_i, def_i) \mid 1 \leq i \leq n\}$. C is empty for all DM nodes (i.e. they are leaf nodes in strategy trees).

Definition 4.2.4 (DM Group Node)**.** Groups can be executed in two ways: sequentially or parallel. The way they are executed is defined by two additional parameters:

- **parallel** $= (\texttt{parallel}, \{\textbf{true}, \textbf{false}\}, \textbf{false})$ for specifying whether this is a sequential or a parallel group
- **dm_evaluator** $= (\texttt{dm\_evaluator}, Str, \texttt{default\_evaluator\_dm})$ is needed only in parallel groups to determine the best result

A DM group node is a DM strategy node $(\epsilon, \{\textbf{time}, \textbf{runs}, \textbf{parallel}, \textbf{dp\_evaluator}\}, PA, C)$ where the list C contains at least one child node.

Definition 4.2.5 (Preprocessing Strategy Tree)**.** A preprocessing strategy tree is a tree that is rooted by some DM strategy node.

Executing Strategy Trees for Preprocessing

Execution of strategy trees for preprocessing is almost identical to the execution of DP-analysis strategy trees (See Section 4.1). Here, we describe the differences that arise for the execution of preprocessing strategy trees. The most important difference is, that now the execution function is not a mapping from sets of DP problems to sets of DP problems but a mapping from TRSs to TRSs. The execution of DM group nodes works exactly analogously to the execution of DP group nodes. For the execution of DM nodes, the only difference to the execution of DP processor nodes is, that now the input is not a set of DP problems but rather a single TRS. Therefore, we can restrict ourselves to the base case, where only one input is available.

The Strategy Language

The syntax for the strategy specification is based on XML. The user-specified strategy is passed to VMTL (CLI) as a ".xml"-file. Syntactic validity of the given file is (to a certain degree) automatically checked by VMTL by validating it against an XML-schema-definition (See Section 4.2 for details).

A valid XML-file specifying a VMTL-strategy has to contain exactly one strategy element. The strategy element can specify zero or one preprocessing elements and zero or one DP analysis elements. The preprocessing and DP analysis elements have to have exactly one child element, which can be either a group element or an element representing a direct method (for descendants of the preprocessing element) or a DP processor (for descendants of the DP analysis element, respectively). A group element can contain a sequence of other group nodes or elements representing direct methods resp. DP processors. The sequence has to contain at least one element, but other than that can contain arbitrarily many elements. The parameters of group nodes resp. direct method- or DP processor nodes are present as attributes of the respective element. If a certain attribute is missing, the default value is assumed. This way, by choosing good default values, it is possible to make the input file less "verbose". Another advantage of using XML as basis for the strategy specification is that parsing is very easy due to the many XML-parsers available. Also, XML-schema-definitions (see below) allow to easily check for syntactic and logical errors.

The exact syntax for the strategy can be read off the schema definition in Section 4.2. As an example for a valid strategy, the XML representation of the default strategy is provided in the Appendix (A.1).

XML-Schema-Definitions

An XML-schema-definition (XSD) is an XML-based definition of a class of XML documents. An XML document is said to validate against an XSD, if it is in the class defined by that XSD. XML parsers can check whether an XML document validates against a certain XSD. If validation fails, they can produce human-readable output describing why it failed. This way, syntactic errors in an XML document can easily be located.

Due to the modular nature of VMTL, it is not possible to use a static XSD, because each DP processor and each direct method has to be considered. Therefore, VMTL compiles the XSD each time it has to validate a given strategy specification. All information about the parameters provided by DP processors and direct methods is available at runtime through their implemented VMTL-API interfaces. This information is used to compile the XSD.

The basic type, on which groups as well as DP processors and direct methods are based is a node. A node is an abstract type, that cannot be instantiated directly, but it declares the attribute "`time`" for the maximum execution time of a node, as well as the attribute "`runs`" for the number of runs of this node. Both attributes are of type "nonNegativeInteger".

```
1 <!-- abstract type TANode;
2      Group and DP-Processor- and DM-nodes extend this -->
3 <xsd:complexType name="TANode">
```

```
4  <xsd:attribute name="time" type="xsd:nonNegativeInteger" default="0"/>
5  <xsd:attribute name="runs" type="xsd:nonNegativeInteger" default="1"/>
6 </xsd:complexType>
```

Listing 4.1: XSD: TANode (abstract type for nodes)

Groups are another abstract type that extend nodes. Groups provide an additional parameter "`parallel`" (of type boolean) to decide, whether the child elements of the group should be executed in parallel or sequentially. This type is used as base for the two different kinds of groups - direct method groups and DP processor groups. Direct method groups have to specify a parameter "`problem_selector_function_dm`" (of type string). This parameter specifies a method that is used to choose the most simple solution among a given set of rewrite systems if more than one solution is found. Analogously, DP processor groups have to specify a parameter "`problem_selector_function_dp`" (also of type string), to choose the most simple solution among a given set of DP problems. Moreover, both kinds of groups have a sequence of child nodes which are either groups of the corresponding type, or DP processor nodes resp. direct method nodes. For every DP processor and direct method, VMTL will automatically generate the necessary entries in the code below.

```
1   <!-- abstract type TAGroup -->
2   <xsd:complexType name="TAGroup">
3     <xsd:complexContent>
4       <xsd:extension base="TANode">
5         <xsd:attribute name="parallel" type="xsd:boolean" default="false" />
6       </xsd:extension>
7     </xsd:complexContent>
8   </xsd:complexType>
9   <!-- type TGroupPreproc -->
10  <xsd:complexType name="TGroupPreproc">
11    <xsd:complexContent>
12      <xsd:extension base="TAGroup">
13        <xsd:choice minOccurs="1" maxOccurs="unbounded">
14          <xsd:element name="Group" type="TGroupPreproc" />
15          <!-- __GENERATE_DM_ENTRIES__ -->
16        </xsd:choice>
17        <!-- __GENERATE_DM_EVALUATOR_ATTRIBUTE -->
18      </xsd:extension>
19    </xsd:complexContent>
20  </xsd:complexType>
21  <!-- type TGroupDPAnalysis -->
22  <xsd:complexType name="TGroupDPAnalysis">
23    <xsd:complexContent>
24      <xsd:extension base="TAGroup">
25        <xsd:choice minOccurs="1" maxOccurs="unbounded">
26          <xsd:element name="Group" type="TGroupDPAnalysis" />
27          <!-- __GENERATE_DP_ENTRIES__ -->
28        </xsd:choice>
29        <!-- __GENERATE_DP_EVALUATOR_ATTRIBUTE -->
30      </xsd:extension>
31    </xsd:complexContent>
32  </xsd:complexType>
```

Listing 4.2: XSD: TGroup (type for Group nodes)

The root node of an XML document defining a VMTL strategy is a "Strategy" element. This element can have a "Preprocessing" element and a "DP analysis" element as children. Both of these can be either a group of the corresponding type or a single DP processor resp. a single direct method. The associated type is defined below. Again, VMTL automatically generates all necessary entries for DP processors and direct methods.

```
1  <!-- Type definition for the strategy node -->
2   <xsd:complexType name="TStrategy">
3     <xsd:sequence>
4       <xsd:element name="Preprocessing"
5                    type="TPreprocessing"
6                    minOccurs="0" maxOccurs="1" />
7       <xsd:element name="DP-Analysis"
8                    type="TDP-Analysis"
9                    minOccurs="0" maxOccurs="1"  />
10    </xsd:sequence>
11  </xsd:complexType>
12  <xsd:complexType name="TPreprocessing">
13    <xsd:choice minOccurs="1" maxOccurs="1">
14      <xsd:element name="Group" type="TGroupPreproc" />
15      <!-- __GENERATE_DM_ENTRIES__ -->
16    </xsd:choice>
17  </xsd:complexType>
18  <xsd:complexType name="TDP-Analysis">
19    <xsd:choice minOccurs="1" maxOccurs="1">
20      <xsd:element name="Group" type="TGroupDPAnalysis" />
21      <!-- __GENERATE_DP_ENTRIES__ -->
22    </xsd:choice>
23  </xsd:complexType>
```

Listing 4.3: XSD: Strategy element and type

For each direct method and DP processor, the associated types have to be defined. For each one of those entities, VMTL includes the following code to the XSD. `_NAME_` is replaced by the Java class-name of the entity. The parameters are extracted from the class through the interface, and the attributes are defined accordingly.

```
1 <!-- template for direct methods and DP-Processors -->
2 <xsd:complexType name="T_NAME_">
3  <xsd:complexContent>
4    <xsd:extension base="TANode">
5      <!-- definition of additional attributes -->
6    </xsd:extension>
7  </xsd:complexContent>
8 </xsd:complexType>
```

Listing 4.4: XSD: Template for DP processors and direct methods

Example 4.2.6. Consider the strategy, described in Example 4.1.6. A strategy file specifying this strategy for VMTL could look as follows:

```
<Strategy>
  <DP-Analysis>
    <Group>
      <DependencyGraph/>
      <Group parallel="true">
        <ReductionPairSAT time="10" runs="3"/>
        <KBOProcessor time="10" runs="3" />
      </Group>
    </Group>
  </DP-Analysis>
</Strategy>
```

Listing 4.5: Example Strategy

As an example of a fully compiled XSD, the XSD generated by VMTL with all the default DP processors and direct methods is included in the appendix in Section A.2.

4.3 The VMTL Source Code

In this section, we provide a quick overview of the organization of the VMTL-source code, that may be helpful when implementing new methods and processors. For a detailed description of the classes presented here, we refer to the VMTL-Javadoc documentation. Basically, the source code consists of three projects:

DeepVisDS
: Contains the core data structures used in VMTL.

DeepVis
: Implementation of the VMTL program logic (uses DeepVisDS).

DeepVisCLI
: Implements the VMTL CLI (uses DeepVis and DeepVisDS).

We now consider the projects in more detail and give an overview of the most important classes.

DeepVisDS

This project defines the most important data structures that are used in VMTL. The interfaces in the package `dpvis.datastructures` provide the core data-types that are used to represent terms, rules, term rewriting systems and dependency pair problems for VMTL. The implemented versions of the data-types defined in this package can be found in the package `dpvis.datastructures.impl`.

The DeepVisDS-project also provides the base classes for all entities that can be built to extend VMTL (e.g. direct methods or DP processors). The base-classes for direct methods can be found in the package `dpvis.datastructures.directMethod`. The class `DirectMethod` has to be used as a base class for all new direct methods and the class `CSDirectMethod` (which extends `DirectMethod`), has to be used as the base class for all context-sensitive direct methods. Similarly, the package `dpProcessor` from `dpvis.datastructures` defines the classes `DPProcessor` (for implementing context-free DP processors) and `ContextSensitiveDpProcessor` (for implementing context-sensitive DP processors).

DeepVis

This project implements the core functionality of VMTL. It contains all currently implemented direct methods and DP processors as well as the methods for parsing the input-files and the strategy, executing strategies, output generation, etc.

The set of currently implemented direct methods can be found in the VMTL package `dpvis.logic.dm.methods` and the set of currently implemented DP processors can be found in the VMTL package `dpvis.logic.dp`.

This project also contains the VMTL Sat Solving Facility (VSSF) which can be found in the package `vssf.core` (for the core functionality) resp. `vssf.factories` (for the factory classes). A detailed overview of the VSSF can be found in Chapter 5.

DeepVisCLI

This project uses the classes defined in the DeepVis- and DeepVisDS-projects to execute a proof search. It consists of just one class "`Main`" in the package `dpvis.cli.main` which implements the CLI-version of VMTL.

CHAPTER 5

The VMTL Sat Solving Facility

In this chapter we introduce the *VMTL SAT Solving Facility* (VSSF). The VSSF provides a generic Java front-end to various SAT- and SMT solvers. It allows for a flexible specification of SAT and SMT problems and encapsulates the invocation of the different solvers on these problems and the interpretation and presentation of their results in a unified way. It is designed to be simple and convenient to use, yet still powerful and flexible. Much of the complexity is hidden from the user. However, if it is necessary, it allows a very flexible configuration of its program logic. One other very important aspect about the VSSF is extensibility. It is designed for easy extension by new solvers and theories for SMT solving.

This chapter starts with a very brief introduction to SAT solving and SMT solving. In Section 2, we are going to introduce the formalism we use for problem specification. Section 3 contains an overview of the classes provided by the VSSF and shows how they interact and how the VSSF is used to specify SAT and SMT problems, solve these problems and use the obtained models. Section 4 introduces the so-called formula classes for easier specification of SAT and SMT formulas and Section 5 provides a tutorial on how to solve some concrete tasks with the VSSF. The description of the VSSF provided in this chapter is just an overview and by no means complete. For a complete specification of all classes, we refer to the API documentation.

5.1 Introduction

SAT solving is a decision problem where one is interested in whether a given formula in (classical) propositional logic is satisfiable or not. The SAT problem is NP-complete ([Coo71]). However, today many very efficient SAT solvers exist (e.g. Minisat [SE05], CLASP [GKNS07], RSAT [PD07], SATzilla [XHHLB08], etc...). Most SAT-solvers use a variant of the DPLL (Davis-Putnam-Logemann-Loveland) procedure ([DLL62]) in order to determine, whether a SAT problem is satisfiable or not. In addition to just determining, whether a given formula is satisfiable, all modern SAT solvers can also present the user a satisfying variable assignment (called a *model*) if satisfiability is detected.

SMT solving is an extension of SAT solving to special versions of first-order logic where certain theories are assumed (i.e. some functions and predicates are fixed). SMT problems can be expressed using these theories. Since theories are not limited to finite domains, SMT solving is more powerful than SAT solving. SMT solvers try to efficiently decide satisfiability of SMT problems by incorporating their knowledge about their supported theories in the search for a model. One example of a theory that an SMT solver could provide may be relations and linear arithmetic on integers.

Example 5.1.1. Let x_1, x_2 and x_3 be integer-variables and let $>$ be the greater-relation on integers and $+$ the addition operation on integers. The following formula is an example of an SMT-Problem:

$$x_1 > x_2 + x_3$$

A model for this problem would be the assignment where $x_1 = 5$, $x_2 = 2$, $x_3 = 2$.

The VSSF provides a front-end to different SAT and SMT solvers. One of the most important decisions in the design of the VSSF was to make it as transparent to the user as possible. The VSSF provides the most commonly used theories that are used in many SMT solvers and defines their semantics within the VSSF-framework. Every solver (SAT or SMT) that is integrated in the VSSF declares which ones of the provided theories it supports. Problems can be specified using the theories provided by the VSSF. If a solver supports all the theories used in some problem specification, it can be used to solve this problem. From the point of view of the user, it does not matter which solver is used (except for performance considerations...). Each solver, that supports all the theories used in a problem specification, yields the same result according to the defined semantics of the theories. Even tough SAT solvers do not support theories directly, SAT solvers implemented in the VSSF can support theories. Suppose some VSSF encapsulation of a SAT solver supports some theory T. When a problem specification using theory T is passed to the SAT solver, VMTL transparently transforms the problem specification into an equivalent one that does not use the theory T, by encoding the expressions from the theory T that were used in the original specification into propositional logic. One theory, for which the VSSF provides automatic transformation into propositional logic for SAT solvers is the theory of bit-vectors.

A similar amount of transparency is provided when extending the VSSF by new solvers or theories. For example, if a new SAT solver should be implemented, VSSF already provides classes that can handle automatic generation of DIMACS (cf. [Dim93], the specification language used in SAT-competitions) input based on a VSSF problem specification and interpretation of the DIMACS output. Also for SAT solvers, the transformation of supported theories is done automatically. Therefore, the integration of a new SAT solver into the VSSF can be done very efficiently and just amounts to writing code that invokes the SAT solver tool with the generated input and passes its output back to the DIMACS interpreter which then extracts the finished model.

5.2 The Specification Logics

Problem instances that can be solved with the VSSF are specified in a set of special logics that generalize propositional logic by allowing the use of predicates (with fixed semantics) over elements from arbitrary domains. The predicates that are available for expressing formulas depends on the theories that are used. This section introduces this kind of logics which we call $\mathcal{T}$-logics. First, we define the abstract notion of a theory for our setting.

Definition 5.2.1. A *theory* $\mathcal{T}$ consists of:

- A domain D (some arbitrary set of elements over which this theory reasons).
- A set F of function symbols with arities $\mathtt{ar}_F : F \mapsto \mathbb{N}_0$.
- A set P of predicate symbols with arities $\mathtt{ar}_P : P \mapsto \mathbb{N}_0$.
- For each $f \in F$, a function $f_{\mathcal{T}} : D^{\mathtt{ar}_F(f)} \mapsto D$.
- For each $p \in P$, a function $p_{\mathcal{T}} : D^{\mathtt{ar}_P(p)} \mapsto \{\mathbf{true}, \mathbf{false}\}$.

Concisely, a theory can be specified as a tuple $(D, F, P, \{f_{\mathcal{T}} \mid f \in F\}, \{p_{\mathcal{T}} \mid p \in P\})$.

Example 5.2.2. A simple theory about integers that allows addition and multiplication as well as basic comparisons could be defined as

$$\mathsf{INT} = (\underbrace{\mathbb{N}_0}_{D}, \underbrace{\{+, \times\}}_{F}, \underbrace{\{=, >\}}_{P}, \{+_{\mathsf{INT}}, \times_{\mathsf{INT}}\}, \{=_{\mathsf{INT}}, >_{\mathsf{INT}}\})$$

where $\mathtt{ar}_F(+) = \mathtt{ar}_F(\times) = 2$ and $\mathtt{ar}_P(=) = \mathtt{ar}_P(>) = 2$. Furthermore, we define the functions $x_1 +_{\mathsf{INT}} x_2 = x_1 + x_2$ and $x_1 \times_{\mathsf{INT}} x_2 = x_1 * x_2$ for all $x_1, x_2 \in D$, where $+$ and $*$ are the addition and multiplication operations over natural numbers. For the predicates $=$ and $>$, we define for all $x_1, x_2 \in D$:

$$x_1 =_{\mathsf{INT}} x_2 = \begin{cases} \mathbf{true} & \text{if } x_1 = x_2 \\ \mathbf{false} & \text{otw.} \end{cases} \quad \text{and} \quad x_1 >_{\mathsf{INT}} x_2 = \begin{cases} \mathbf{true} & \text{if } x_1 > x_2 \\ \mathbf{false} & \text{otw.} \end{cases}$$

where $=$ and $>$ denote the respective relations over the natural numbers.

Definition 5.2.3. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be theories where $\mathcal{T}_n = (D_n, F_n, P_n, \{f_{\mathcal{T}_n} \mid f \in F_n\}, \{p_{\mathcal{T}_n} \mid p \in P_n\})$ $(n \in \{1, 2\})$. $\mathcal{T}_1$ and $\mathcal{T}_2$ are called *disjoint* iff the sets F_1 and F_2 are disjoint and the sets P_1 and P_2 are disjoint. A set $\boldsymbol{\mathcal{T}} = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ ot theories is called a *set of pairwise disjoint theories*, if for all $i, j \in \{1, \ldots, n\}$ with $i \neq j$, $\mathcal{T}_i$ and $\mathcal{T}_j$ are disjoint.

Let $\boldsymbol{\mathcal{T}}$ be a set of pairwise disjoint theories. The $\boldsymbol{\mathcal{T}}$-logic generalizes propositional logic in the sense, that instead of propositional variables, one may also use predicates from any of the theories in $\boldsymbol{\mathcal{T}}$. As arguments for the used predicates, elements from the respective domain of the theory or functions over them may be used.

The following two sections formally establish syntax and semantics of $\boldsymbol{\mathcal{T}}$-logics as well of the notion of satisfiability of formulas specified in these logics.

Syntax

The syntax of $\boldsymbol{\mathcal{T}}$-logics is defined in a way similar to how the syntax of first order logic is usually defined. However, the set of predicate symbols and function symbols is fixed by the set of theories $\boldsymbol{\mathcal{T}}$ and a very basic form of typing is used, as predicate symbols and function symbols permit only elements from the domain of the respective theory as their arguments. Furthermore, $\boldsymbol{\mathcal{T}}$-logic formulas may not contain quantifiers.

Concerning variables, we define a countable infinite set V_p of propositional variables and for each theory $\mathcal{T}_i \in \boldsymbol{\mathcal{T}}$ over the domain D_i, we choose a countable infinite set of variables V_{D_i} as well. We require all of the defined sets to be pairwise disjoint from each other. Later, when defining the semantics, variables may only be assigned values from the adequate domain (i.e. for variables from V_p, we permit $\mathbf{true}$ and $\mathbf{false}$ and for variables in the set V_{D_i}, we permit values from D_i.

Definition 5.2.4 ($\mathcal{T}$-Terms)**.** Let $\mathcal{T}$ be a theory with domain D and function symbols F. The set of $\mathcal{T}$*-terms* is defined inductively to be the smallest set such that

- all elements $a \in D$ are $\mathcal{T}$-terms,
- all variables $v \in V_D$ are $\mathcal{T}$-terms and
- for all $f \in F$, $f(t_1, \ldots, t_{\mathtt{ar}_F(f)})$ is a $\mathcal{T}$-term if $t_1, \ldots, t_{\mathtt{ar}_F(f)}$ are $\mathcal{T}$-terms.

Terms that are elements from D are called *constant terms* and terms that are elements from V_D are called *variable terms*. Constant terms and variable terms are also called *atomic terms*.

A $\mathcal{T}$-term is called *ground* if it does not contain any variables.

Definition 5.2.5 ($\boldsymbol{\mathcal{T}}$-Formulas)**.** Let $\boldsymbol{\mathcal{T}}$ be a set of pairwise disjoint theories. The set of $\boldsymbol{\mathcal{T}}$-formulas is defined inductively to be the smallest set such that

- $\top$ and $\bot$ are $\boldsymbol{\mathcal{T}}$-formula
- all propositional variables $v \in V_p$ are $\boldsymbol{\mathcal{T}}$-formulas
- let $\mathcal{T}_i \in \boldsymbol{\mathcal{T}}$ be a theory with predicates P_i. If p is some predicate from P_i and $t_1, \ldots, t_{\mathtt{ar}_{P_i}(p)}$ are $\mathcal{T}_i$-terms, then $p(t_1, \ldots, t_{\mathtt{ar}_{P_i}(p)})$ is a $\boldsymbol{\mathcal{T}}$-formula
- if f_1 and f_2 are $\boldsymbol{\mathcal{T}}$-formulas, then
  - $\neg f_1$ and $\neg f_2$ are $\boldsymbol{\mathcal{T}}$-formulas and
  - $f_1 \circ f_2$ is a $\boldsymbol{\mathcal{T}}$-formula for all $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$

We say, that a $\boldsymbol{\mathcal{T}}$-formula is *closed* iff it does not contain any variables, otherwise it is called *open*. This is similar to first order logic, where a formula is called closed iff it does not contain any free variables. Since $\boldsymbol{\mathcal{T}}$-logics do not support quantifiers, all variables occurring in a $\boldsymbol{\mathcal{T}}$-formula are free.

Example 5.2.6. For $\boldsymbol{\mathcal{T}} = \emptyset$, we obtain the $\emptyset$-logic that corresponds to classical propositional logic.

Example 5.2.7. For $\boldsymbol{\mathcal{T}} = \{\mathsf{INT}\}$, where INT is the theory defined in Example 5.2.2, we obtain the $\{\mathsf{INT}\}$-logic that allows expressing relations over natural numbers. Suppose, $p_1 \in V_p$ is a propositional variable and $i_1, i_2 \in V_{\mathbb{N}_0}$ are integer variables. Then the following formula is a valid formula in the $\{\mathsf{INT}\}$-logic:

$$((i_1 = 5) \rightarrow p_1) \wedge (p_1 \leftrightarrow (i_2 > (i_1 + 6)))$$

In the next subsection, we define the semantics of the formulas defined in this subsection.

Semantics

Semantics of $\boldsymbol{\mathcal{T}}$-logics are defined by means of valuation functions that compute the truth value of any given $\boldsymbol{\mathcal{T}}$-formula. Before we can define the valuation functions, we must take care of the variables occurring in open formulas. To this end, we define $\boldsymbol{\mathcal{T}}$-variable assignments, that map each variable to some appropriate value.

Definition 5.2.8 ($\boldsymbol{\mathcal{T}}$-variable assignment $\boldsymbol{\alpha}$)**.** Let $\boldsymbol{\mathcal{T}} = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ be a set of pairwise disjoint theories where the domain of each $T_i \in \boldsymbol{\mathcal{T}}$ is D_i. A $\boldsymbol{\mathcal{T}}$-variable assignment $\boldsymbol{\alpha}$ is a set of functions $\{\alpha_p, \alpha_{D_1}, \ldots, \alpha_{D_n}\}$ such that α_p is a mapping $\alpha_p : V_p \mapsto \{\mathbf{true}, \mathbf{false}\}$ and for each $i \in \{1, \ldots, n\}$, α_{D_i} is a mapping $\alpha_{D_i} : V_{D_i} \mapsto D_i$.

Next, we discuss how $\mathcal{T}$-terms are evaluated under some $\boldsymbol{\mathcal{T}}$-variable assignment $\boldsymbol{\alpha}$. The evaluation of a $\mathcal{T}$-term interprets the term under the given variable assignment. To this end, variables are replaced by the respective values assigned by the variable assignment. The functions specified by the respective theory are then used to evaluate the terms.

Definition 5.2.9 (Evaluation of $\mathcal{T}$-terms)**.** Let $\mathcal{T}$ be some theory over the domain D and let α_D be a mapping $\alpha_D : V_D \mapsto D$. The function $[\cdot]^{\mathcal{T}}_{\alpha_D}$ maps $\mathcal{T}$-terms to D: For all $\mathcal{T}$-terms t we define:

$$[t]^{\mathcal{T}}_{\alpha_D} = \begin{cases} a & \text{if } t = a \text{ and } a \in D \\ \alpha_D(t) & \text{if } t \in V_D \\ f_{\mathcal{T}}([t_1]^{\mathcal{T}}_{\alpha_D}, \ldots, [t_n]^{\mathcal{T}}_{\alpha_D}) & \text{if } t = f(t_1, \ldots, t_n) \text{ for } f \in F \text{ and } t_1, \ldots, t_n \mathcal{T}\text{-terms} \end{cases}$$

Now we are ready to define the valuation function for $\boldsymbol{\mathcal{T}}$-formulas.

Definition 5.2.10. Let $\boldsymbol{\mathcal{T}}$ be a set of pairwise disjoint theories and let $\boldsymbol{\alpha}$ be a $\boldsymbol{\mathcal{T}}$-variable assignment. The mapping $\mathtt{val}^{\boldsymbol{\mathcal{T}}}_{\boldsymbol{\alpha}}$ maps $\boldsymbol{\mathcal{T}}$-formulas to $\{\mathbf{true}, \mathbf{false}\}$. For each $\boldsymbol{\mathcal{T}}$-formula, we define $\mathtt{val}^{\boldsymbol{\mathcal{T}}}_{\boldsymbol{\alpha}}$ inductively as follows:

- $\mathtt{val}^{\boldsymbol{\mathcal{T}}}_{\boldsymbol{\alpha}}(\top) = \mathbf{true}$ and $\mathtt{val}^{\boldsymbol{\mathcal{T}}}_{\boldsymbol{\alpha}}(\bot) = \mathbf{false}$
- $\mathtt{val}^{\boldsymbol{\mathcal{T}}}_{\boldsymbol{\alpha}}(v) = \alpha_p(v)$ for all propositional variables $v \in V_p$

- $\mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(p(t_1, \ldots, t_n)) = p_{\mathcal{T}_i}([t_1]^{\mathcal{T}_i}_{\alpha_{D_i}}, \ldots, [t_n]^{\mathcal{T}_i}_{\alpha_{D_i}})$ for all predicate symbols $p \in P_i$ with P_i being the set of predicate symbols from some theory $\mathcal{T}_i \in \boldsymbol{\mathcal{T}}$ and $\mathcal{T}_i$-terms t_1 to t_n
- $\mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(\neg F) = \begin{cases} \mathbf{true} & \text{if } \mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F) = \mathbf{false} \\ \mathbf{false} & \text{if } \mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F) = \mathbf{true} \end{cases}$
- $\mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F_1 \wedge F_2) = \begin{cases} \mathbf{true} & \text{if } \mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F_1) = \mathbf{true} \text{ and } \mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F_2) = \mathbf{true} \\ \mathbf{false} & \text{otw.} \end{cases}$
- $\mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F_1 \vee F_2) = \begin{cases} \mathbf{true} & \text{if } \mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F_1) = \mathbf{true} \text{ or } \mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F_2) = \mathbf{true} \\ \mathbf{false} & \text{otw.} \end{cases}$
- $\mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F_1 \rightarrow F_2) = \begin{cases} \mathbf{true} & \text{if } \mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F_2) = \mathbf{true} \text{ whenever } \mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F_1) = \mathbf{true} \\ \mathbf{false} & \text{otw.} \end{cases}$
- $\mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F_1 \leftrightarrow F_2) = \begin{cases} \mathbf{true} & \text{if } \mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F_1) = \mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F_2) \\ \mathbf{false} & \text{otw.} \end{cases}$
- $\mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F_1 \oplus F_2) = \begin{cases} \mathbf{true} & \text{if } \mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F_1) \neq \mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F_2) \\ \mathbf{false} & \text{otw.} \end{cases}$

Example 5.2.11. Consider the following $\emptyset$-formula F:

$$(v_1 \wedge v_2) \rightarrow (v_1 \vee v_2)$$

Let $\boldsymbol{\alpha} = \{\alpha_p\}$ be a $\emptyset$-variable assignment, where $\alpha_p(v_1) = \mathbf{true}$ and $\alpha_p(v_2) = \mathbf{false}$. The valuation of the formula F is: $\mathtt{val}^{\emptyset}_{\boldsymbol{\alpha}}((v_1 \wedge v_2) \rightarrow (v_1 \vee v_2))$. This evaluates to $\mathbf{true}$, if whenever $\mathtt{val}^{\emptyset}_{\boldsymbol{\alpha}}(v_1 \wedge v_2)$ evaluates to $\mathbf{true}$, also $\mathtt{val}^{\emptyset}_{\boldsymbol{\alpha}}(v_1 \vee v_2)$ evaluates to $\mathbf{true}$. $\mathtt{val}^{\emptyset}_{\boldsymbol{\alpha}}(v_1 \wedge v_2)$ evaluates to $\mathbf{true}$ only if $\mathtt{val}^{\emptyset}_{\boldsymbol{\alpha}}(v_1)$ and $\mathtt{val}^{\emptyset}_{\boldsymbol{\alpha}}(v_2)$ both evaluate to $\mathbf{true}$. Both expressions are base cases and $\mathtt{val}^{\emptyset}_{\boldsymbol{\alpha}}(v_1) = \mathbf{true}$ but $\mathtt{val}^{\emptyset}_{\boldsymbol{\alpha}}(v_1) = \mathbf{false}$, so we know $\mathtt{val}^{\emptyset}_{\boldsymbol{\alpha}}(v_1 \wedge v_2) = \mathbf{false}$ and thus $\mathtt{val}^{\emptyset}_{\boldsymbol{\alpha}}((v_1 \wedge v_2) \rightarrow (v_1 \vee v_2)) = \mathbf{true}$

Example 5.2.12. Let $\boldsymbol{\mathcal{T}} = \{\mathsf{INT}\}$. Consider the $\{\mathsf{INT}\}$-formula F from Example 5.2.7:

$$((i_1 = 5) \rightarrow p_1) \wedge (p_1 \leftrightarrow (i_2 > (i_1 + 6)))$$

Let $\boldsymbol{\alpha} = \{\alpha_p, \alpha_{\mathbb{N}_0}\}$ be an $\{\mathsf{INT}\}$-variable assignment, where $\alpha_p(p_1) = \mathbf{true}$ and $\alpha_{\mathbb{N}_0}(i_1) = 5$ and $\alpha_{\mathbb{N}_0}(i_2) = 2$. The valuation of the formula is

$$\mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(((i_1 = 5) \rightarrow p_1) \wedge (p_1 \leftrightarrow (i_2 > (i_1 + 6))))$$

We have to determine $\mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}((i_1 = 5) \rightarrow p_1)$ and $\mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(p_1 \leftrightarrow (i_2 > (i_1 + 6)))$. In order to determine the value of $\mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}((i_1 = 5) \rightarrow p_1)$, we first check $\mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(i_1 = 5)$. This evaluates to $\mathbf{true}$ because it becomes $[i_1]^{\mathcal{T}}_{\alpha_{\mathbb{N}_0}} =_{\mathsf{INT}} [5]^{\mathcal{T}}_{\alpha_{\mathbb{N}_0}}$ and since $\alpha_{\mathbb{N}_0}(i_1) = 5$ this is $\mathbf{true}$. Now, since $\mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(p_1)$ is also $\mathbf{true}$, we know that $\mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}((i_1 = 5) \rightarrow p_1) = \mathbf{true}$. In a similar way, we obtain $\mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(p_1 \leftrightarrow (i_2 > (i_1 + 6))) = \mathbf{false}$ and thus $\mathtt{val}^{\mathcal{T}}_{\boldsymbol{\alpha}}(F) = \mathbf{false}$.

Satisfiability of T-Logic Formulas

In this short section, we present the notion of satisfiability of $\mathcal{T}$-formulas. Satisfiability of $\mathcal{T}$-formulas F is defined through the existence of *models* for F. We always require theories to be pairwise disjunct in order to avoid ambiguity due to using the same function name or predicate name twice in different theories.

Definition 5.2.13. Let $\mathcal{T}$ be a set of pairwise disjoint theories and F be some $\mathcal{T}$-formula. Further, let $\boldsymbol{\alpha}$ be a $\mathcal{T}$-variable assignment. $\boldsymbol{\alpha}$ is called a $\mathcal{T}$*-model* of F iff $\mathtt{val}_{\boldsymbol{\alpha}}^{\mathcal{T}}(F) = \mathbf{true}$. If the set of theories used is clear from the context, we will usually just use the term *model*.

Definition 5.2.14. Let $\mathcal{T}$ be a set of pairwise disjoint theories and F be some $\mathcal{T}$-formula. F is called *satisfiable* iff there is a $\mathcal{T}$-model for F. Otherwise, F is called *unsatisfiable*.

Definition 5.2.15. Let $\mathcal{T}$ be a set of pairwise disjoint theories and F be some $\mathcal{T}$-formula. F is called *valid* iff every $\mathcal{T}$-variable assignment $\boldsymbol{\alpha}$ is a $\mathcal{T}$-model for F.

Example 5.2.16. Consider the $\emptyset-$formula F

$$(v_1 \wedge v_2) \rightarrow (v_1 \vee v_2)$$

and the $\emptyset$-variable assignment $\boldsymbol{\alpha}$ from Example 5.2.11. As we have shown before, $\mathtt{val}_{\boldsymbol{\alpha}}^{\emptyset}((v_1 \wedge v_2) \rightarrow (v_1 \vee v_2)) = \mathbf{true}$, thus we have a model for F and so F is satisfiable. Furthermore, as one can easily check (there are only four interesting variable assignments that need to be checked), F is also valid, because *every* $\emptyset$-variable assignment is a model for F.

Example 5.2.17. Consider the $\{\mathsf{INT}\}$-formula F from Example 5.2.7:

$$((i_1 = 5) \rightarrow p_1) \wedge (p_1 \leftrightarrow (i_2 > (i_1 + 6)))$$

As shown in Example 5.2.12, F is not valid because an $\{\mathsf{INT}\}$-variable assignment is given that makes F $\mathbf{false}$. However, F is still satisfiable. Take for example the $\{\mathsf{INT}\}$-variable assignment $\boldsymbol{\alpha} = \{\alpha_p, \alpha_{\mathbb{N}_0}\}$ where $\alpha_p(p_1) = \mathbf{true}$ and $\alpha_{\mathbb{N}_0}(i_1) = 5$ and $\alpha_{\mathbb{N}_0}(i_2) = 12$. It is easily verified, that $\mathtt{val}_{\boldsymbol{\alpha}}^{\mathcal{T}}(F) = \mathbf{true}$.

Example 5.2.18. As an example for a $\{\mathsf{INT}\}$-formula that is unsatisfiable, take the following formula F:

$$(i_1 = 1) \wedge (i_1 = 2)$$

Where i_1 is a variable from the set $V_{\mathbb{N}_0}$. Obviously, there can be no assignment to i_1 that makes F $\mathbf{true}$, becasue it would have assign 1 and 2 to i_1 at the same time.

5.3 Built-In Theories of the VSSF

The VSSF offers a selection of useful theories that are supported by most SMT solvers. In this section, we specify the available functions and predicates for each of the built-in theories and fix their semantics.

BitVectors

Bit-vectors are very basic, yet very flexible and versatile entities that can be used in problem specifications.

Definition 5.3.1 (Bit-Vector)**.** Let $B = \{0, 1\}$ be the set of bit-values. A tuple $\langle b_n, b_{n-1}, \ldots, b_1 \rangle$ ($b_1, \ldots, b_n \in B$) is called a bit-vector of length n (or n-bit bit-vector). By B^n we denote the set of all n-bit bit-vectors. Bit-vectors constitute binary representations of integer values where the bit b_n is the most valuable bit and the bit b_1 is the least-valuable bit. The representation can be either signed or unsigned. In the signed case, we assume that the encoding is in two's complement notation. The functions $\texttt{int}_u : B^n \mapsto \mathbb{N}_0$ and $\texttt{int}_s : B^n \mapsto \mathbb{Z}$ (for all $n \geq 1$) map bit-vectors to their unsigned / signed integer values.

Notation. For noting bit-vectors, we will usually use the C-style notation for binary numbers. For example, instead of $\langle 0, 1, 1, 0, 0, 1 \rangle$, we will write `0b011001`.

Example 5.3.2. The bit-vector `0b011001` has the following unsigned and signed integer values: $\texttt{int}_u(\texttt{0b011001}) = 25$ and $\texttt{int}_s(\texttt{0b011001}) = 25$. The bit-vector `0b111001` has the following integer values: $\texttt{int}_u(\texttt{0b111001}) = 57$ and $\texttt{int}_s(\texttt{0b111001}) = -7$

The BitVectors Theory

The theory $\mathsf{BitVectors}_n$ is defined on the domain B^n and provides the following predicates and functions:

Functions:

- Arithmetic functions:

Function	Arity	Description
$+$	2	Addition (modulo 2^n)
$-$	2	Subtraction (modulo 2^n)

- Bit-operations:

Function	Arity	Description
$<<_0$	1	left-shift by 1 bit (introducing a 0 as the new lvb)
$<<_1$	1	left-shift by 1 bit (introducing a 1 as the new lvb)
$>>_0$	1	right-shift by 1 bit (introducing a 0 as the new mvb)
$>>_1$	1	right-shift by 1 bit (introducing a 1 as the new mvb)
`neg`	1	negates all bits
`and`	2	bitwise and
`or`	2	bitwise or
`impl`	2	bitwise implication
`equiv`	2	bitwise equivalence
`xor`	2	bitwise xor

Predicates:

Predicate	Arity	Description
$>$	2	greater
$>_s$	2	signed greater
$=$	2	equality

The semantics of the defined function symbols is the obvious intended semantics. For the predicates, we define:

$$bv_1 =_{\mathsf{BitVectors_n}} bv_2 = \begin{cases} \mathbf{true} & \text{if } bv_1 \text{ and } bv_2 \text{ coincide at all bit-positions} \\ \mathbf{false} & \text{otw.} \end{cases}$$

$$bv_1 >_{\mathsf{BitVectors_n}} bv_2 = \begin{cases} \mathbf{true} & \text{if } \mathtt{int}_u(bv_1) > \mathtt{int}_u(bv_2) \\ \mathbf{false} & \text{otw.} \end{cases}$$

$$bv_1 >_{s\mathsf{BitVectors_n}} bv_2 = \begin{cases} \mathbf{true} & \text{if } \mathtt{int}_s(bv_1) > \mathtt{int}_s(bv_2) \\ \mathbf{false} & \text{otw.} \end{cases}$$

Example 5.3.3. An example for a formula in the $\{\mathsf{BitVectors_5}\}$-logic is:

$$bv_1 = (bv_2 + 0b00110) \wedge \neg((bv_2 \,\mathtt{and}\, bv_2) = 0b00000)$$

where bv_1 and bv_2 are bit-vector variables from the set V_{B^5}. A satisfying variable assignment for this formula is: $bv_1 = \mathtt{0b11110}$ and $bv_2 = \mathtt{0b11000}$

Integers

The theory of integers (lnts) allows to specify natural operations and relations on integer numbers. The domain of this theory is $\mathbb{Z}$ and the following functions and predicates are provided:

Functions:

Function	Arity	Description
$+$	2	Addition
$-$	2	Subtraction
$*$	2	Multiplication
`div`	2	Integer-division
`mod`	2	Remainder

Predicates:

Predicate	Arity	Description
$>$	2	greater
$=$	2	equality
`positive`	1	positivity
`negative`	1	negativity

The semantics of the defined functions and predicates correspond to the semantics of the respective functions and relations over the integer numbers. The predicate `positive` is **true** for positive numbers and **false** otherwise and the predicate `negative` is **true** for negative numbers and **false** otherwise.

Reals

The theory of rational numbers (called Reals for consistency with SMT-LIB and with Java) allows to specify natural operations and relations on rational numbers. The domain of this theory is $\mathbb{Q}$. The following functions and predicates are provided:

Functions:

Function	Arity	Description
$+$	2	Addition
$-$	2	Subtraction
$*$	2	Multiplication
$\div$	2	Division
`div`	2	Integer-division
`mod`	2	Remainder

Predicates:

Predicate	Arity	Description
$>$	2	greater
$=$	2	equality
`positive`	1	positivity
`negative`	1	negativity

The semantics of the defined functions and predicates correspond to the semantics of the respective functions and relations over the real numbers.

5.4 Overview of the VSSF

This section provides an overview of the classes provided by the VSSF and shows how they interact. This section is intended as an overview of the functionality of the VSSF. For a more in-depth resource on the classes provided by the VSSF, we refer to the VSSF documentation.

Basically, the VSSF consists of three parts that interact with each other:

1. Problem Specification
2. Solvers
3. Models

The problem specification part is the biggest part of VMTL. It allows for the specification of SAT and SMT problems. The specified problems are then passed to a solver which tries to find a model. The obtained model can finally be used to interpret the variables occurring in the problem specification.

Problem Specification

Here, we introduce the classes that are used to specify problem instances which can be solved by the VSSF. We distinguish between two types of problems: clause problems and formula problems. A clause problem consists of a set of clauses that should be satisfied and a formula problem consists of a set of formulas that should be satisfied (called assertions).

Clause Problems

Most SAT solvers take as input not propositional formulas but sets of clauses. The VSSF offers this very basic way to specify SAT problems.

Definition 5.4.1 (Literal)**.** A *literal* is either some propositional variable p or the negation of some propositional variable (i.e. $\neg p$).

Definition 5.4.2 (Clause)**.** A *clause* is a set of literals. A clause corresponds to the disjunction of its literals, i.e. a clause is satisfied iff at least one of its literals evaluates to $\mathbf{true}$.

Definition 5.4.3 (Clause Problem)**.** A clause problem is a set of clauses. A clause problem is called satisfied if all the clauses it contains are satisfied. Thus, a clause problem corresponds to the conjunction of all its clauses.

Example 5.4.4. Let p_1, p_2, p_3 be propositional variables. Then the following is a clause problem:

$$\big\{\{p_1, p_2, \neg p_3\}, \{p_2, p_3\}, \{\neg p_1, \neg p_2, \neg p_3\}\big\}$$

This clause problem is satisfied by the variable assignment where $p_1 = \mathbf{true}$, $p_2 = \mathbf{false}$ and $p_3 = \mathbf{true}$. The clause problem corresponds to the following propositional formula:

$$(p_1 \vee p_2 \vee \neg p_3) \wedge (p_2 \vee p_3) \wedge (\neg p_1 \vee \neg p_2 \vee \neg p_3)$$

Propositional variables are represented by the class `PVariable` in the VSSF. If a new propositional variable is needed, one just creates a new instance of `PVariable`.

Literals are created as instances of the class `Literal`. The constructor of this class takes as arguments the propositional variable and the polarity of the created literal. Clauses are represented by the class `Clause`. After instantiating a new `Clause`-object, the method `addLiteral` can be used to add the literals.

Clause-problems are represented by the class `ClauseProblem`. After instantiating a new clause problem, the method `addClause` can be used to add clauses.

Definition 5.4.5 (Subsumption)**.** A clause c is subsumed by another clause c' iff c is a superset of c'. In a clause problem, clauses that are subsumed by some other clause in the clause problem can be removed because if the smaller clause is satisfied, the bigger clause is automatically satisfied as well.

The removal of subsumed clauses offers a very basic form of optimizing clause problems. However, in practice subsumption-checking is rather expensive ($O(n^2)$). Therefore, the VSSF does not automatically remove subsumed clauses. The method `doSubsumption` of the `ClauseProblem` class can be used to remove all subsumed clauses from the set of clauses contained in the clause problem.

Implementation Detail
In the VSSF, subsumption is done by first sorting the clauses by their size and then checking for set-inclusion of the smaller sets in the bigger sets, removing the bigger sets if subsumption is detected. This yields quadratic runtime ($O(n^2/2)$ to be exact).

Example 5.4.6. Consider as an example the clause problem from Example 5.4.4:

$$\big\{\{p_1, p_2, \neg p_3\}, \{p_2, p_3\}, \{\neg p_1, \neg p_2, \neg p_3\}\big\}$$

This clause problem is specified in the VSSF by the following code:

```
1  ClauseProblem p = new ClauseProblem();
2
3  PVariable p1 = new PVariable();
4  PVariable p2 = new PVariable();
5  PVariable p3 = new PVariable();
6
7  Literal lp1  = new Literal(p1, Polarity.positive);
8  Literal lnp1 = new Literal(p1, Polarity.negative);
9  Literal lp2  = new Literal(p2, Polarity.positive);
10 Literal lnp2 = new Literal(p2, Polarity.negative);
11 Literal lp3  = new Literal(p3, Polarity.positive);
12 Literal lnp3 = new Literal(p3, Polarity.negative);
13
14 Clause c1 = new Clause();
15 Clause c2 = new Clause();
16 Clause c3 = new Clause();
17
18 c1.addLiteral(lp1);
19 c1.addLiteral(lp2);
20 c1.addLiteral(lnp3);
21
22 c2.addLiteral(lp2);
23 c2.addLiteral(lp3);
24
25 c3.addLiteral(lnp1);
26 c3.addLiteral(lnp2);
27 c3.addLiteral(lnp3);
```

```
28
29 p.addClause(c1);
30 p.addClause(c2);
31 p.addClause(c3);
```

Listing 5.1: Specifying a simple clause problem

As the example demonstrates, specifying SAT problems in this way s not very convenient. Therefore, the VSSF offers different, more sophisticated ways of specifying problem instances.

Formula Problems

A more natural way of specifying SAT problems (and the only way to specify SMT problems...) are formula problems. Here, problems can be specified by building $\mathcal{T}$-formulas and adding them to the problem specification.

We first show, how propositional formulas are specified in the VSSF and how they are added to a formula problem. SAT solvers usually expect clause sets as input, therefore, we show how the VSSF translates propositional formulas into clause sets. After that, we show how the theories defined in Section 5.3 are integrated in the VSSF and how they are used.

Specifying Formulas over the $\emptyset$-Logic (i.e. Propositional Logic)

If no theories are used, we obtain raw propositional formulas. These formulas can be constructed using the usual connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ and $\oplus$ as well as propositional variables that are used as atoms. The semantics is the usual semantics of the corresponding connectives as defined in Section 5.2. The following classes are used to specify a formula that represents the corresponding connective. All classes shown here are based on the class `Formula`, so by Java's polymorphism, they can be used everywhere, where a `Formula`-instance is expected.

Formula	Class	Constructor Arguments
$\top$	`Formula.Verum`	-
$\bot$	`Formula.Falsum`	-
p (atom)	`Formula.Atom`	`PVariable` or *TheoryPredicate*
$\neg$	`Formula.Neg`	*Formula*
$\wedge$	`Formula.And`	*Formula*, *Formula*
$\vee$	`Formula.Or`	*Formula*, *Formula*
$\rightarrow$	`Formula.Impl`	*Formula*, *Formula*
$\leftrightarrow$	`Formula.Equiv`	*Formula*, *Formula*
$\oplus$	`Formula.Antiv`	*Formula*, *Formula*

The constructors of these classes take as arguments zero, one or two other formulas (resp. atomic entities for atoms): zero for the logical constants, one for atoms and for negation and two for the other connectives, according to their usual arity. This naturally captures the inductive definition of propositional formulas. For now, we are only concerned with formulas where all atoms are

propositional variables. In the next subsection, we discuss how predicates from some theory can be used to build SMT formulas.

Example 5.4.7. Consider the formula corresponding to the clause set from Example 5.4.4:

$$(p_1 \vee p_2 \vee \neg p_3) \wedge (p_2 \vee p_3) \wedge (\neg p_1 \vee \neg p_2 \vee \neg p_3)$$

A specification containing this formula can be created as follows in the VSSF:

```
1  Problem p = new Problem();
2
3  PVariable p1 = new PVariable();
4  PVariable p2 = new PVariable();
5  PVariable p3 = new PVariable();
6
7  Formula f = new And(new Or(new Or(new Atom(p1),
8                                    new Atom(p2)),
9                                    new Neg(new Atom(p3))),
10             new And(new Or(       new Atom(p2),
11                                   new Atom(p3)),
12                       new Or(new Or(new Neg(new Atom(p1)),
13                                   new Neg(new Atom(p2))),
14                                   new Neg(new Atom(p3)))));
15
16 p.addAssertion(f);
```

Listing 5.2: Specifying a simple formula problem

In Section 5.6, we will introduce the `FormulaFactory`-class which allows even shorter specifications of formulas.

Implementation Detail
Note that, although everything is declared with the **`new`**-keyword, the VSSF is smart enough to realize, that two different instances of the same formula have the same meaning, so in optimizations it can handle them accordingly. The VSSF is even smart enough to realize, that for commutative formulas like `And`, `Or`, `Equiv`, etc..., the order of the argument formulas does not matter, so if two instances of such a formula have the same argument formulas, just with swapped position, they are still considered equivalent.

Specifying Formulas Over Theories

Theories are created in the VSSF as classes extending the abstract *`Theory`* class. Theory classes in the VSSF are a collection of inner classes that make up the Theory:

- One class representing the variables for this theory (extends the *`TheoryVariable`* class).

- A class that represents variable-terms.
- A class that represents constant-terms.
- For each function symbol a class representing the corresponding function (extending the *Term* class).
- For each predicate symbol of the theory a class representing that predicate (extending the `Predicate` class).

Implementation Detail
Type safety is enforced by the VSSF by the use of Java's Generics concept. Thus, most abstract base classes used in conjunction with theories are parametrized by some type parameters. The result is, that it is not possible to accidentally use terms from a different theory in predicates of another theory because Java's static program analysis already prevents this. This makes the VSSF quite consistent in the sense of how formulas are specified and minimizes the need for runtime-exception checking. When specifying formulas, the use of generics is mostly hidden and is not required from the user. However, if terms should be stored in a collection, it is convenient to use the common base-class which is typed. For this case, we will state the actual type of the base-class of terms for every theory when presenting the theory. It can also be looked up in the VSSF-documentation.

Theory predicates are classes that are derived from the class *TheoryPredicate*. The constructors of theory predicates take as arguments instances of terms over the respective theory.

In order to build terms for some theory, every theory provides its own type of variables (derived from the class *TheoryVariable*) and its own constant type, which can be any Java type (e.g. Java's `Integer` class for the Ints-theory or a special type `BitVector` for the BitVectors theory).

Formulas may also contain predicates over theories. As mentioned before, they may be used in the place of propositional variables in formulas. Formulas using such theory predicates can be built using the same classes that are used to build propositional formulas, but now the class `Atom` may be built from some instance of the *TheoryPredicate* class.

In the following subsections, we show how the built-in theories of the VSSF (defined in Section 5.3) are implemented in the VSSF. Note, that the implemented versions may contain additional features (not defined in Section 5.3) that are added for the sake of comfort. These features do not fit our theoretical framework for defining theories (extending the framework to actually capture everything the VSSF implementations of the theories are capable of, would make the framework very bloated and messy and hard to understand, therefore, we just state the additional comfort features here and in the VSSF documentation).

Bit-Vectors

The theories $\mathsf{BitVectors}_n$ ($n \geq 1$) are implemented as the class `TheoryBitVectors`.

> **Implementation Detail**
> Note, that we provide only one class that implements all of the $\mathsf{BitVectors}_n$ ($n \geq 0$) theories. The VSSF checks at runtime, that all operations receive as input bit-vector terms of the same length. In the case, that it is tried to perform some operation with differently sized bit-vector terms, a runtime exception is thrown. Sadly, no static check of correctness is possible in this case; This is a limitation of the Java programming language, where static class parameters may only be types, but not instances of types (i.e. no integer values).

The variable-type is `TheoryBitVectors.BVVariable` (or just `BVVariable`). Instances of this type represent bit-vector variables that can be used in bit-vector terms. When a new `BVVariable` instance is created, the desired bit-size of the variable is specified. Derived from the `BVVariable`-class is the `BVVariableSigned`-class that represents signed bit-vector variables.

The constant-type is `BitVector`. This class provided by the VSSF stores bit-vectors and can perform basic operations on bit-vectors (see the VSSF-documentation for details). When a new `BitVector` is instantiated, the size as well as the desired value can be provided. It is also possible to just provide the value, in which case the smallest possible size is automatically chosen. If both the size and the value are provided, a runtime exception (`InvalidSizeException`) is thrown, if the desired value does not fit the specified size. Derived from `BitVector` is `SignedBitVector` which represents the signed version (regarding comparison and integer-interpretation) of `BitVector`. Negative values are stored in two's complement form.

The common base-class of all `TheoryBitVectors`-terms is *`Term`*`<BitVector, BVVariable>` (we abbreviate this as `BVTerm` which is not really a class name but rather a short-hand notation). The following classes are provided to build `TheoryBitVectors`-terms:

Class	Operation	Constructor Arguments
`ConstantTerm`	-	`BitVector`
`VariableTerm`	-	`BVVariable`
`Add`	$+$	`BVTerm`, `BVTerm`
`AddNO`	Non-overflowing addition (*)	`BVTerm`, `BVTerm`
`Sub`	$-$	`BVTerm`, `BVTerm`
`SubNU`	Non-underflowing subtraction (*)	`BVTerm`, `BVTerm`
`Shl0`	$<<_0$ (by k bit) (*)	`BVTerm`, **`int`** ($= k$)
`Shl1`	$<<_1$ (by k bit) (*)	`BVTerm`, **`int`** ($= k$)
`Shr0`	$>>_0$ (by k bit) (*)	`BVTerm`, **`int`** ($= k$)
`Shr1`	$>>_1$ (by k bit) (*)	`BVTerm`, **`int`** ($= k$)
`PTimes`	$\times$ (*)	`BVTerm`, `PVariable`
`Neg`	`neg`	`BVTerm`
`And`	`and`	`BVTerm`, `BVTerm`
`Or`	`or`	`BVTerm`, `BVTerm`
`Impl`	`impl`	`BVTerm`, `BVTerm`
`Equiv`	`equiv`	`BVTerm`, `BVTerm`
`Xor`	`xor`	`BVTerm`, `BVTerm`

Operations marked with (*) are new or extended as compared to the definition of the basic BitVectors$_n$-theories in Section 5.3. We explain their meaning here:

For the shift operations, we allow specifying the number k of bits that should be shifted. This makes the use of the theory more convenient, as one does not have to cascade shift-terms just to shift by more than one bit.

The classes `AddNO` and `SubNU` are convenient ways of expressing, that the respective operations should not overflow (resp. underflow).

Example 5.4.8. Consider the formula

$$0b1011 = bv_1 +_{no} bv_2$$

where $+_{no}$ means non-overflowing addition and bv_1 and bv_2 are bit-vector variables. This formula is only satisfiable, if there are value assignments for the bit-vector variables bv_1 and bv_2 such that their sum is $0b1011$ but no overflow happens in the addition. For example, a model of this formula would be $bv_1 = 0b1010$ and $bv_2 = 0b0001$. But $bv_1 = 0b1110$ and $bv_2 = 0b1101$ is *no* model for this formula. However, if regular, overflowing addition would have been used it would be a model because $\texttt{int}_u(0b1110) = 14$, $\texttt{int}_u(0b1101) = 13$ and $14 + 13 = 27$ and $27 \; mod \; 2^4 = 11 = \texttt{int}_u(0b1011)$.

The $\times$-operation is another new operation that may be useful in some cases. It allows "multiplication" of a bit-vector bv with a single bit p (i.e. a propositional variable). If p is **true**, then $bv \times p$ is bv, otherwise the result is a bit-vector of the same length as bv where all bits are set to **false**.

The following predicate classes are defined by the `BitVector` class. They represent the corresponding predicates from Section 5.3.

Class	Predicate	Constructor Arguments
`Greater`	$>$	`BVTerm`, `BVTerm`
`Equal`	$=$	`BVTerm`, `BVTerm`

Implementation Detail
The operations do not distinguish between working with signed or unsigned bit-vectors. The only difference between `BitVector` and `SignedBitVector` resp. `BVVariable` and `BVVariableSigned` is that their interpretation function decodes them correctly as signed / unsigned bit-vectors into integers of the corresponding value. Furthermore, the `Greater` predicate respects the signedness.

Example 5.4.9. Consider the following formula (from Example 5.3.3):

$$bv_1 = (bv_2 + 0b00110) \wedge \neg((bv_1 \,\texttt{and}\, bv_2) = 0b00000)$$

Suppose, bv_1, bv_1 and bv_1 are signed bit-vector variables. The formula can be specified in the VSSF using the following code:

```
1  Problem p = new Problem();
2
3  BVVariable bv1 = new BVVariable();
4  BVVariable bv2 = new BVVariable();
5
6  Formula f =
7    new And(new Atom(new Equals(new VariableTerm(bv1),
8                             new Add(new VariableTerm(bv2),
9                                     new ConstantTerm(new BitVector(5,6))))),
10           new Neg(new Atom(new Equals(new TheoryBitVectors.And(
11                                                 new VariableTerm(bv1),
12                                                 new VariableTerm(bv2)),
13                                       new ConstantTerm(new BitVector(5,0))))))
14   )
15 p.addAssertion(f);
```

Listing 5.3: Specifying a simple problem using bit-vectors

Again, in Section 5.5, we will introduce a factory class that reduces the size of the required code a little.

Ints

The theory Ints is implemented in the VSSF as the class `TheoryInts`. Variables are of the type `IntsVariable` and constants are of the type `Integer` (the regular Java integers). The common type of terms is: *Term*<`Integer, IntsVariable`> (which we will abbreviate by `IntsTerm` here).

The following classes are provided by `TheoryInts` to build terms:

Class	Operation	Constructor Arguments
ConstantTerm	-	Integer
VariableTerm	-	IntsVariable
Add	+	IntsTerm, IntsTerm
Sub	−	IntsTerm, IntsTerm
Mul	*	IntsTerm, IntsTerm
IntDiv	div	IntsTerm, IntsTerm
Mod	mod	IntsTerm, IntsTerm

In order to express predicates, the following classes are provided by TheoryInts:

Class	Predicate	Constructor Arguments
Greater	>	IntsTerm, IntsTerm
Equal	=	IntsTerm, IntsTerm
Positive	positive	IntsTerm
Negative	negative	IntsTerm

Reals

The theory Reals is implemented in the VSSF as the class TheoryReals. Variables are of the type RealsVariable and constants are of the type Double (the Java class for floating-point values). The common type of terms is: *Term*<Double, RealsVariable> (which we will abbreviate by RealsTerm here).

The following classes are provided by TheoryReals to build terms:

Class	Operation	Constructor Arguments
ConstantTerm	-	Real
VariableTerm	-	RealsVariable
Add	+	RealsTerm, RealsTerm
Sub	−	RealsTerm, RealsTerm
Mul	*	RealsTerm, RealsTerm
Div	÷	RealsTerm, RealsTerm
IntDiv	div	RealsTerm, RealsTerm
Mod	mod	RealsTerm, RealsTerm

In order to express predicates, the following classes are provided by TheoryReals:

Class	Predicate	Constructor Arguments
Greater	>	RealsTerm, RealsTerm
Equal	=	RealsTerm, RealsTerm
Positive	positive	RealsTerm
Negative	negative	RealsTerm

This concludes the section about problem specifications for the VSSF. In the next section, we discuss how the problems specified by the means presented in this section are solved by the VSSF solvers.

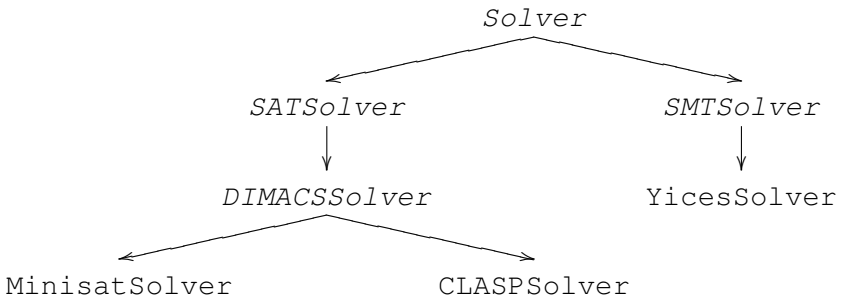
Solving

All solvers that are implemented in the VSSF extend the *Solver* base class. This abstract class declares the following functions, which concrete solver classes may implement:

- `isSat`: Takes as input an instance of `Problem` or `ClauseProblem` and returns **`true`** if the model is satisfiable or **`false`** if it is not satisfiable.
- `getModel`: Takes as input an instance of `Problem` or `ClauseProblem` and returns a model if the problem is satisfiable. If it is not satisfiable, **`null`** is returned.
- `getModels`: Takes as input an instance of `Problem` or `ClauseProblem` as well as a number $n \geq 1$. Returns a set of up to n models. If less than n models exist, only the existing models are returned. If no model exists, an empty set is returned.
- `getAllModels`: Takes as input an instance of `Problem` or `ClauseProblem` and returns the set of all models for the problem. If no model exists, an empty set is returned.

Of the four methods, only the method `isSat` is mandatory for concrete solver classes to implement. However, all currently provided solvers also implement at least the `getModel` method. If an unimplemented method is invoked for some concrete solver class, a runtime exception (viz. `OperationNotSupportedException`) is thrown.

The following diagram depicts the class-hierarchy of the currently provided solvers and their intermediate helper classes which perform tasks like input generation and output parsing for the output of the underlying solvers. Class names that are written in italics denote abstract classes, which cannot be instantiated directly.



The solver classes perform the following tasks:

1. Bring the `Problem` resp. `ClauseProblem` instances they receive as input into some form that the underlying solver can understand.
2. Invoke the underlying SAT or SMT solver using the generated, translated problem specification as input.

3. Wait for the underlying solver to finish, then collect its output.

4. Parse the output generated by the underlying solver and create a VSSF model (an instance of the `Model` class) and return it. (Or if the user did not request a model (i.e. called `isSat`) just return `true` or `false` accordingly).

Converting Formula Problems to Clause Problems and Vice Versa

As mentioned earlier, the underlying SAT- and SMT solvers expect their input in different formats. Most SAT solvers expect their input in DIMACS format while SMT solvers usually provide a proprietary input specification language (although SMT-LIB Version 1.2 [RT06] or SMT-LIB Version 2.0 [BST+10] are supported by many SMT solvers).

The SMT solvers usually provide a rich input specification language, so formula problems can be translated into the format expected by the solvers very naturally. However, clause problems have to be converted to formula problems before SMT solvers can solve them. This is done automatically by the VSSF base classes in the obvious straightforward way (as shown in Example 5.4.4).

The other direction requires a little more effort: SAT solvers require their input in the form of clause sets, so formula problems, which may also contain theories, must be converted to SAT-equivalent clause problems. This is done in a two step process:

1. Predicates from theories that can be translated into propositional logic are translated. If other predicates that cannot be translated to propositional logic are encountered, a runtime exception is thrown.

2. The now theory-free formulas are translated into propositional logic.

The translations are done automatically by the VSSF framework, but the default settings may be changed if necessary. Translations are only required for SAT solvers, so the abstract *`SATSolver`* class implements this functionality. The first step (translating theory predicates into propositional logic) is done by so-called theory transformer classes (classes derived from the abstract base class *`TheoryTransformer`*. The second step is done by a formula translator (classes derived from the abstract base class *`FormulaTranslator`*). In the following we discuss the two steps in a little more detail.

The *`TheoryTransformer`* class declares two abstract methods. One method that translates problems containing predicates from some theory to problems not containing such predicates by translating them into propositional logic. The other method takes as input a model for the translated problem and builds from it a model for the original problem. This second method is discussed in more detail in Section 5.4. Actual theory transformers have to implement both methods. Currently, the only theory that supports automatic translation to propositional logic is the `BitVectors` theory. The corresponding theory transformer is called `BitVectorsTransformer`. The transformation is done by means of bit-blasting.

The second step involved in translating formula problems into clause problems is the actual translation of formulas into clauses. The abstract base class *`FormulaTranslator`* declares

the method that is used for the translation. This method takes as input a set of formulas and returns a set of clauses. Every concrete formula translator extending the *FormulaTranslator* class must adhere the following rules:

- The set of clauses that is returned is satisfiable, if and only if the original formula is satisfiable (SAT equivalence).
- The set of clauses that is returned can contain extra variables. Every model of the set of clauses, restricted to the variables of the original formula, is also a model of the original formula (model compatibility)

Implementation Detail
The *FormulaTranslator* class provides a caching mechanism to its subclasses. Therefore, (sub-)formulas that have already been translated once need not be translated again.

The VSSF implements two different translation techniques: CNF-translation and Tseitin translation. The CNF translator is implemented in the `CNFTranslator` class. This transformation does not introduce fresh propositional variables. It follows the standard procedure of transforming the formula into conjunctive normal form. The conjunctive normal form can then directly be used to obtain clause sets. This method has exponential worst-case runtime. Because of this, it is only suitable for the translation of rather small formulas.

The other translation technique is based on Tseitin-style translations [Tse68]. This translation creates a clause form that is only SAT-equivalent, but not fully logically equivalent to the original formula. It works by introducing new variables for each sub-formula. It is enforced, that each variable is satisfiable if and only if the corresponding sub-formula is satisfiable. The VSSF implementation of the Tseitin translation also supports the Plaisted-Greenbaum extension [PG86], where the if and only if condition can be weakened, and only the required direction is enforced to hold (based on the polarity of the formula). The Tseitin translator is implemented in the class `TseitinTranslator`. The Tseitin translation has linear runtime and is well suited for translating big formulas. Using the Plaisted-Greenbaum extension yields even fewer clauses.

Implementation Detail
In the VSSF, each variable is declared either as a `user`-variable or as an `aux`-variable. All variables created by the user are automatically declared as `user`-variables. Variables that are created by VSSF in the process of some transformation (like the Tseitin translation) are always declared as `aux`-variables. The `aux`-variables are basically hidden from the user. They do not appear in models and they cannot (and usually need not and should not) be referenced, because this would create a dependency on a certain concrete translation process which should be avoided.

Both steps involved in transforming a formula problem into a clause problem are automatically handled by the *SATSolver* class, which is a base class of all implemented SAT solvers. To this end, the *SATSolver* class contains a list of *TheoryTransformer* instances. All of these are applied sequentially as the first step. By default, one translator is contained, namely a BitVectorsTransformer. For the second step, it contains a *FormulaTranslator* instance. By default the formula translator is a TseitinTranslator instance where the Plaisted-Greenbaum extension is enabled. The configured *FormulaTranslator* is applied repeatedly for each formula in the formula problem. The *SATSolver* class also provides the following methods to override the default settings:

addTheoryTransformer Takes as input an instance of a *TheoryTransformer* and adds it to the set of theory transformers that are applied first, when trying to solve a formula problem with the SAT solver.

clearTheoryTransformers Removes all the theory transformers that are currently in the set of theory transformers of this SAT solver.

setFormulaTranslator Takes as input an instance of a *FormulaTranslator* and makes it the new formula translator that is used when trying to solve formula problems with the SAT solver.

Example 5.4.10. Solving SMT problems is very easy with the VSSF. It suffices to instantiate some solver and pass the problem to it.

```
1 Problem p = new Problem();
2 /* ... (construct some formula problem p) */
3
4 Solver s = new YicesSolver(); // instantiates a new SMT solver (Yices)
5
6 Model m = s.getModel(p);
```

Listing 5.4: Solving formula problems with an SMT solver

We will discuss how to work with the obtained model in the next section.

Example 5.4.11. Working with SAT solvers is just as easy. Suppose, some formula p is constructed containing only predicates from the BitVectors theory.

```
1 Problem p = new Problem();
2 /* ... (construct some formula problem p using just the BitVectors theory) */
3
4 Solver s = new MinisatSolver(); // instantiates a new SAT solver (Minisat)
5
6 Model m = s.getModel(p);
```

Listing 5.5: Solving formula problems with a SAT solver

All necessary transformations happen automatically in the background. They can also be changed. Suppose, instead of the preconfigured Tseitin translation with enabled Plaisted-Greenbaum extensions we want to use a Tseitin translation without the extension. This is done by the following code:

```
1  Problem p = new Problem();
2  /* ... (construct some formula problem p using just the BitVectors theory) */
3
4  Solver s = new MinisatSolver(); // instantiates a new SAT solver (Minisat)
5
6  TseitinTranslator ttl = new TseitinTranslator();
7  ttl.usePlaistedGreenbaumExtension(false);
8  s.setFormulaTranslator(ttl);
9
10 Model m = s.getModel(p);
```

Listing 5.6: Configuring a SAT solver to use a different clause translation process

Models

The final important part of the VSSF are the models obtained by the solvers. Basically, a model is a variable assignment for all variables of a problem, such that the problem is satisfied. Models are instances of the `Model` class. This class provides the method `getVariableAssignment` which takes as input a variable and returns the value of this variable under the model. The type of the returned value depends on the constant type of the corresponding theory that introduced the variable. For propositional variables (instances of `PVariable`), a boolean value is returned. For bit-vector variables (`BVVariable`), a bit-vector (`BitVector`) is returned. For the Ints variables and Reals variables, `Integer` resp. `Double` values are returned.

> **Implementation Detail**
> The type of the returned value of the `getVariableAssignment` method is determined with the help of the Java Generics feature at compile time. Therefore, no cast is necessary and type safety is maintained.

Example 5.4.12. Consider the formula we specified in Example 5.4.7:

$$(p_1 \vee p_2 \vee \neg p_3) \wedge (p_2 \vee p_3) \wedge (\neg p_1 \vee \neg p_2 \vee \neg p_3)$$

We now extend the code from Example 5.4.7 and let Java print the determined variable assignment:

```
1  Problem p = new Problem();
2
3  PVariable p1 = new PVariable();
4  PVariable p2 = new PVariable();
5  PVariable p3 = new PVariable();
6
7  Formula f = new And(new Or(new Or(new Atom(p1),
8                                    new Atom(p2)),
9                                    new Neg(new Atom(p3))),
10             new And(new Or(       new Atom(p2),
```

```
11                                  new Atom(p3)),
12                      new Or(new Or(new Neg(new Atom(p1)),
13                                  new Neg(new Atom(p2))),
14                                  new Neg(new Atom(p3)))));
15
16 p.addAssertion(f);
17
18 Solver s = new MinisatSolver();
19 Model m = s.getModel(p);
20
21 if(m == null)
22 {
23    System.out.println("UNSAT");
24 }
25 else
26 {
27    System.out.println("p1: " + m.getVariableAssignment(p1));
28    System.out.println("p2: " + m.getVariableAssignment(p2));
29    System.out.println("p3: " + m.getVariableAssignment(p3));
30 }
```

Listing 5.7: Specifying and solving a simple formula problem

Executing this code yields the following output (other solvers may find a different model):

```
p1: false
p2: true
p3: true
```

In addition to the `Solver.getVariableAssignment` method, the VSSF offers a different very convenient way of interpreting all kinds of entities that contain variables. The VSSF provides an interface called `Interpretable`. The interface provides a method `interpret` that takes as input a model and returns the value of the corresponding entity under the model. Every specification entity (like all kinds of variables, formulas, terms, and clauses) implements this interface. This allows evaluation of complete formulas or terms under a model.

Example 5.4.13. Consider the following formula using the bit-vectors theory:

$$(bv_1 = bv_2 + bv_3) \oplus (bv_2 > bv_1)$$

The following codes looks for a model of this formula and then interprets both sides of the antivalence connective, to see which one is true under the found model. It also prints the interpretations of the bit-vector variables both as bit-vectors and in integer representation.

```
1 Problem p = new Problem();
2
3 BVVariable bv1 = new BVVariable(5);
4 BVVariable bv2 = new BVVariable(5);
5 BVVariable bv3 = new BVVariable(5);
6
7 Formula f1 = new Atom(new Equals(new VariableTerm(bv1),
```

```
8                                   new AddNO(new VariableTerm(bv2),
9                                             new VariableTerm(bv3))));
10 Formula f2 = new Atom(new Greater(new VariableTerm(bv2),
11                                    new VariableTerm(bv1)));
12 Formula f = new Antiv(f1, f2);
13 p.addAssertion(f);
14
15 SATSolver sv = new MinisatSolver();
16 Model m = sv.getModel(p);
17
18 if(m == null)
19    System.out.println("UNSAT");
20 else
21 {
22    System.out.println("bv1: " + bv1.interpret(m) +
23          "(" + bv1.interpret(m).getIntValue() + ")");
24    System.out.println("bv2: " + bv2.interpret(m) +
25          "(" + bv2.interpret(m).getIntValue() + ")");
26    System.out.println("bv3: " + bv3.interpret(m) +
27          "(" + bv3.interpret(m).getIntValue() + ")");
28
29    System.out.println("formula1: " + f1.interpret(m));
30    System.out.println("formula2: " + f2.interpret(m));
31    System.out.println("formula1 xor formula2: " + f.interpret(m));
32 }
```

Listing 5.8: Using the "Interpretable"-interface

A possible output is:

```
bv1: 00000(0)
bv2: 00000(0)
bv3: 00000(0)
first disjunct: true
second disjunct: false
formula1 xor formula2: true
```

5.5 The Factory Classes

The VSSF provides many classes, many of which share equal names (e.g. `Neg` for negation in propositional formulas, negation of bit-vectors, ...). Therefore, usually, qualified class-names have to be used (e.g. `Formula.Neg` resp. `TheoryBitVectors.Neg`, ...), if more theories are used that provide classes with the same name. This is no problem for automatically generated formulas, but when specifying formulas by hand, it may get tedious.

Another problem is that in Java every instance of a class has to be constructed with the help of the **`new`**-keyword. This also makes specifying formulas quite verbose.

As a more convenient way of specifying formulas are the so-called "factory classes" that are provided by the VSSF. There are factory classes for each theory. Use of the factory classes for

specifying formulas and terms is recommended as it makes the code shorter and more easy to understand.

Example 5.5.1. Suppose, we want to specify the following problem, where bv_1, bv_2 and bv_3 are bit-vector variables, r_1 and r_2 are real variables, and i_1 is an integer variable:

$$(\texttt{0b01100} = bv_2 +_{NO} bv_3) \wedge (bv_1 > bv_2)$$
$$\texttt{positive}(i_1) \oplus (i_1 > 20)$$
$$(r_1 > 22) \wedge (r_2 = r_1 * 41.52)$$

These assertions can be specified using factory classes in the following way:

```
1  Problem p = new Problem();
2
3  BVVariable bv1 = new BVVariable(size);
4  BVVariable bv2 = new BVVariable(size);
5  BVVariable bv3 = new BVVariable(size);
6  IntVariable i1 = new IntVariable();
7  RealVariable r1 = new RealVariable();
8  RealVariable r2 = new RealVariable();
9
10 PropositionalFactory pf  = new PropositionalFactory();
11 BitVectorsFactory    bvf = new BitVectorsFactory();
12 IntsFactory          inf = new IntsFactory();
13 RealsFactory         rf  = new RealsFactory();
14
15 Formula f1 = pf.atom(bvf.equal(bvf.atom(new BitVector(size, 12)),
16                                 bvf.addNO(bvf.atom(bv2), bvf.atom(bv3))));
17 Formula f2 = pf.atom(bvf.greater(bvf.atom(bv1), bvf.atom(bv2)));
18 Formula f3 = pf.xor(pf.atom(inf.positive(inf.variable(i1))),
19                     pf.atom(inf.greater(inf.variable(i1),
20                                         inf.constant(20))));
21 Formula f4 = pf.and(pf.atom(rf.greater(rf.variable(r1), rf.constant(22.0))),
22                     pf.atom(rf.equal(rf.variable(r2),
23                             rf.times(rf.variable(r1), rf.constant(41.52)))));
24
25 p.addAssertion(f1);
26 p.addAssertion(f2);
27 p.addAssertion(f3);
28 p.addAssertion(f4);
```

Listing 5.9: Using the factory classes

In a test-run of the specified code, the VSSF found the following interpretation (using the Yices-SMT solver):

```
i1: 0
r1: 23.0
r2: 954.96
bv1: 11000(24)
bv2: 00110(6)
bv2: 00110(6)
```

5.6 Using the VSSF

This section provides a tutorial on how to use the VSSF to specify different problems, solve the problems and work with models.

Example 5.6.1. As a first example, consider the following formula in propositional logic, for which satisfiability should be checked:

$$(a \rightarrow b) \vee (b \rightarrow a)$$

The first step is to declare all the propositional variables:

```
1 PVariable a = new PVariable();
2 PVariable b = new PVariable();
```

Next, we create a new problem instance to which we will add the formula.

```
1 Problem p = new Problem();
```

The next step is, to add the formula to the problem instance. In order to specify the formula, we use a formula factory, which we declare first. Next, we use the `addAssertion` method of the problem `p` we declared above, to add the formula.

```
1 PropositionalFactory pf  = new PropositionalFactory();
2
3 p.addAssertion(pf.and(pf.impl(pf.atom(a),pf.atom(b)),
4                       pf.impl(pf.atom(b),pf.atom(a))));
```

At this point, the problem specification is finished. We can now use any SAT or SMT solver to solve problem `p` and obtain a model for it. In this example, we are going to use the Minisat-solver.

```
1 Solver sv = new MinisatSolver();
2 Model m = sv.getModel(p);
```

Now, we can check, if the problem is satisfiable, or not. If it is not satisfiable, then our model `m` will be **`null`**. Otherwise, it contains a variable assignment for our two propositional variables a and b, such that the formula specified above evaluates to true.

```
1 if(m == null)
2    System.out.println("UNSAT");
3 else
4 {
5    System.out.println("a: " + a.interpret(m));
6    System.out.println("b: " + b.interpret(m));
7 }
```

Of course, the formula is satisfiable. One possible output would be:

```
a: false
b: false
```

The Solver-Classes may also provide the possibility to look for more than one (or even all) models of a SAT problem. Currently, only the CLASP solver supports this feature. Consider again the code from Example 5.6.1. We now change the code (starting from the line, where the solver is declared) and use the CLASP solver instead, requesting from it four models. It will return up to four models (Actually, there are only two models, so only those are returned...):

```
1  SATSolver sv = new CLASPSolver();
2  Set<Model> mods = sv.getModels(p, 4);
3
4  if(mods.size() == 0)
5     System.out.println("UNSAT");
6  else
7  {
8     int i = 0;
9     for(Model m : mods)
10    {
11       System.out.print("Model " + i++ + ":");
12       System.out.print(" a: " + a.interpret(m));
13       System.out.println(" b: " + b.interpret(m));
14    }
15 }
```

Now, as expected, both models are printed correctly:

```
Model 0: a: true b: true
Model 1: a: false b: false
```

There is a pitfall, however, when using the feature to find more than one model, that needs to be taken care of. Suppose, we want to find models for the formula

$$(a \vee b) \vee \neg(a \wedge b)$$

This formula is specified as a VSSF problem using the following code:

```
1  p.addAssertion(pf.or(pf.or(pf.atom(a), pf.atom(b)),
2                       pf.not(pf.and(pf.atom(a), pf.atom(b)))));
```

If we use CLASP again, to find four models, we get the following unexpected output:

```
Model 0: a: false b: true
Model 1: a: false b: true
Model 2: a: false b: true
Model 3: a: false b: false
```

We got four models but the first three models are the same. However, there should be exactly four different models, that the Solver could find (every possible variable assignment is a model). The problem is, that the *SATSolver* class by default uses the Tseitin translation with enabled Plaisted-Greenbaum extension in order to translate the formula into a clause set. When the

Plaisted-Greenbaum extension is enabled, it can happen, that some of the auxiliary variables introduced by the translation process are not fixed (i.e. it does not matter how the SAT solver assigns them). Thus, the SAT Solver actually produces four distinct models. However, they only differ on some of the auxiliary variables. Therefore, when looking for more than one model it is necessary to disable the Plaisted-Greenbaum extension for the Tseitin translator. To do this, the following code can be used, after declaring the SAT solver:

```
1 SATSolver sv = new CLASPSolver();
2
3 TseitinTranslator tt = new TseitinTranslator();
4 tt.usePlaistedGreenbaumExtension(false);
5 sv.setFormulaTranslator(tt);
6
7 Set<Model> mods = sv.getModels(p, 4);
```

Now, if we run the code again, we get the expected output:

```
Model 0: a: false b: false
Model 1: a: true b: true
Model 2: a: true b: false
Model 3: a: false b: true
```

Example 5.6.2. Suppose, we want to solve the following mathematical riddle (from the internet[1])

> A merchant can place 8 large boxes or 10 small boxes into a carton for shipping. In one shipment, he sent a total of 96 boxes. If there are more large boxes than small boxes, how many cartons did he ship?

We can specify an SMT formula, using the Ints-theory, such that any model that is found will be the solution to the riddle. We need two variables: The variable numBig will hold the number of cartons with big boxes and the variable numSmall will hold the number of cartons with small boxes inside. We have to specify, that the total number of boxes is equal to 96. This can be done with the following formula:

$$\mathsf{numBig} * 8 + \mathsf{numSmall} * 10 = 96$$

Furthermore, we need to specify, that the number of of big boxes sent is greater than the number of small boxes sent:

$$\mathsf{numBig} * 8 > \mathsf{numSmall} * 10$$

The following block of code specifies these formulas as VSSF-formulas and uses the Yices-solver to solve the problem. After a model has been found, the solution is printed.

[1] `http://www.mathwarehouse.com/riddles/math-riddles.php`, accessed on Sept. 04, 2011

```
1  Problem p = new Problem();
2
3  IntVariable numBig   = new IntVariable();
4  IntVariable numSmall = new IntVariable();
5
6  PropositionalFactory pf = new PropositionalFactory();
7  IntsFactory inf  = new IntsFactory();
8
9  Formula f1, f2;
10
11 /* numBig * 8 + numSmall * 10 = 96 */
12 f1 = pf.atom(inf.equal(inf.add(inf.times(inf.atom(numBig), inf.atom(8)),
13                                 inf.times(inf.atom(numSmall), inf.atom(10))),
14                         inf.atom(96)));
15 /* numBig * 8 > numSmall * 10 */
16 f2 = pf.atom(inf.greater(inf.times(inf.atom(numBig), inf.atom(8)),
17                           inf.times(inf.atom(numSmall), inf.atom(10))));
18
19 p.addAssertion(f1);
20 p.addAssertion(f2);
21
22 Solver sv = new YicesSolver();
23 Model m = sv.getModel(p);
24
25 if(m == null)
26    System.out.println("UNSAT");
27 else
28 {
29
30    System.out.println("There are " +
31                  numBig.interpret(m) + " cartons with big boxes and " +
32                  numSmall.interpret(m) + " cartons with small boxes.");
33    System.out.println("Thus, in total " +
34                  (numBig.interpret(m) + numSmall.interpret(m)) +
35                  " cartons were sent.");
36 }
```

Running this code yields the following output, which provides the solution to the riddle:

```
There are 7 cartons with big boxes and 4 cartons with small
boxes.
Thus, in total 11 cartons were sent.
```

CHAPTER 6

Semantic Labeling

Semantic labeling [Zan95] is a method of transforming TRSs into different TRSs that are terminating if and only if the original TRSs are terminating. The most-commonly used methods of proving termination are based on simplification orders, which means that non simply-terminating([KO92]) TRSs (which still are potentially terminating) cannot be shown terminating with these methods. Transforming a TRS into a different TRS by means of semantic labeling might make a TRS that was previously non-simply-terminating orientable via a simplification order, and thus it may be possible to show termination. The first section of this chapter offers a short introduction to semantic labeling and establishes the notations used in the later sections. Section 2 shows a special version of semantic labeling, called *root-labeling* [SM08], which can be used as both a direct method and a DP processor. In Section 3, we will generalize the direct method version of root-labeling to the context-sensitive case, and in Section 4 we do the same for the DP processor.

6.1 Semantic Labeling

Here, we present the semantic labeling transformation from [Zan95] and discuss how it can be generalized to work with context-sensitive rewriting. After that, we extend it into a CS-DP processor. The idea of semantic labeling is to incorporate some information about the intended semantics of a rewrite system directly into the rewrite system, by adding labels to terms. This way, an automatic prover that normally only examines syntactic features of rewrite systems might, in a natural way, be able to show termination more easily using the new additional information.

Semantics of a TRS, Algebras

The semantics of a rewrite system can be described by an $\mathcal{F}$-algebra. An $\mathcal{F}$-algebra assigns a meaning to each constant and each function symbol in $\mathcal{F}$ and defines the elements over which the rewrite system operates.

Definition 6.1.1 ($\mathcal{F}$-Algebra)**.** Let $\mathcal{F}$ be a signature. An *$\mathcal{F}$-Algebra* $\mathcal{A}$ is a pair $(A, \{f_\mathcal{A} \mid f \in \mathcal{F}\})$ where

- A is a set containing arbitrary elements (called the domain of $\mathcal{A}$) and
- for each $f \in \mathcal{F}^{(n)}$ there is a function $f_\mathcal{A} : A^n \mapsto A$ (for all $n \geq 0$).

Example 6.1.2. Let $\mathcal{F} = \{\mathsf{0}, \mathsf{s}, \mathsf{plus}\}$ be a signature where $\mathtt{ar}(\mathsf{0}) = 0$, $\mathtt{ar}(\mathsf{s}) = 1$ and $\mathtt{ar}(\mathsf{plus}) = 2$. One example of an $\mathcal{F}$-Algebra is the algebra $\mathcal{A}$ with the domain $\mathbb{N}_0$ and functions $\mathsf{0}_\mathcal{A} = 0$, $\mathsf{s}_\mathcal{A}(x) = x + 1$ and $\mathsf{plus}_\mathcal{A}(x, y) = x + y$, where 0 is the natural number zero and $+$ denotes addition over natural numbers.

In order to evaluate a term in an $\mathcal{F}$-algebra $\mathcal{A}$, variables occurring in the term need to be mapped to members of the domain A of $\mathcal{A}$. This is done by a *variable assignment* function α.

Definition 6.1.3 (Variable Assignment)**.** For the set X of variables and a domain A, a *variable assignment* α is a function $\alpha : X \mapsto A$. The set of all possible variable assignments is denoted as A^X.

Evaluation of a term $t \in \mathcal{T}(\mathcal{F}, X)$ under an $\mathcal{F}$-algebra $\mathcal{A}$, wrt. a variable assignment α, amounts to replacing variables in the term with their assigned elements from the domain, and replacing the pure syntactic function symbols with their corresponding functions defined by $\mathcal{A}$.

Definition 6.1.4 (Term Evaluation)**.** Let $\mathcal{A}$ be an $\mathcal{F}$-algebra and let α be a variable assignment. A *term evaluation* is a function $[\alpha] : \mathcal{T}(\mathcal{F}, X) \mapsto A$ defined by

$$\begin{array}{rcll} [\alpha](x) & = & \alpha(x) & \text{f.a. } x \in X \\ [\alpha](f(t_1, \ldots, t_n)) & = & f_\mathcal{A}([\alpha](t_1), \ldots, [\alpha](t_n)) & \text{f.a. } f \in \mathcal{F}^{(n)}, t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, X) \end{array}$$

Example 6.1.5. Let $\mathcal{F}$ and $\mathcal{A}$ be as in Example 6.1.2. Let α be a variable assignment whith $\alpha(x) = 2$ and $\alpha(y) = 1$. The evaluation of the term $\mathsf{plus}(x, \mathsf{s}(y))$ under the variable assignment α is:

$$[\alpha](\mathsf{plus}(x, \mathsf{s}(y))) = \mathsf{plus}_\mathcal{A}(2, \mathsf{s}_\mathcal{A}(1)) = 2 + 2 = 4$$

The underlying semantics of term rewriting is, that a rewrite step is intended to be some kind of transition from one term to another, that does only apply syntactic changes to terms, but does not change the intended value the term evaluates to. This is similar to rewriting a mathematical term - for example, instead of $(2 + 2)$ one might write 4. Only the syntactic representation changes but the intended value is in both cases the number 4. An $\mathcal{F}$-algebra that satisfies this property for every rule of a rewrite system $\mathcal{R}$ is called a *model* of $\mathcal{R}$.

Definition 6.1.6 (Model)**.** Let $\mathcal{R}$ be a TRS and let $\mathcal{A}$ be an $\mathcal{F}$-algebra. $\mathcal{A}$ is called a *model for* $\mathcal{R}$ if for each rule $l \to r \in \mathcal{R}$ and for every variable assignment $\alpha \in A^X$, $[\alpha](l)$ and $[\alpha](r)$ evaluate to the same element in A (i.e. $[\alpha](l) = [\alpha](r)$).

Example 6.1.7. Consider the following TRS $\mathcal{R}$ over the signature $\mathcal{F}$ from Example 6.1.2:

$$\begin{array}{lcllcl} \mathsf{plus}(\mathsf{0},\mathsf{x}) & \to & \mathsf{x} & \mathsf{plus}(\mathsf{s}(\mathsf{x}),\mathsf{y}) & \to & \mathsf{s}(\mathsf{plus}(\mathsf{x},\mathsf{y})) \\ \mathsf{plus}(\mathsf{x},\mathsf{0}) & \to & \mathsf{x} & \mathsf{plus}(\mathsf{x},\mathsf{s}(\mathsf{y})) & \to & \mathsf{s}(\mathsf{plus}(\mathsf{x},\mathsf{y})) \end{array}$$

The algebra $\mathcal{A}$ from Example 6.1.2 is a model for $\mathcal{R}$. To see this, consider the rule $\mathsf{plus}(\mathsf{s}(\mathsf{x}),\mathsf{y}) \to \mathsf{s}(\mathsf{plus}(\mathsf{x},\mathsf{y}))$. The left-hand-side evaluates to $1 + x + y$ and the right-hand-side evaluates to $x + y + 1$ (for $x, y \in \mathbb{N}_0$). Obviously, both sides are equivalent for all possible substitutions of x and y into the natural numbers. Using similar argumentation, one can easily prove, that the same also holds for all other rules.

In the following, usually $\mathcal{M}$ (An $\mathcal{F}$-Algebra with domain M and functions $f_{\mathcal{M}}$) denotes a model for some TRS. A model for a TRS $\mathcal{R}$ defines one possible semantics for $\mathcal{R}$.

Semantic Labeling

Semantic labeling uses a model $\mathcal{M}$ and a special *labeling function* to apply labels to function symbols occuring in terms. To this end, for each $n \geq 0$ and each function symbol $f \in \mathcal{F}^{(n)}$, a new non-empty set of labels, denoted as S_f, is chosen. The signature $\mathcal{F}$ is extended to $\overline{\mathcal{F}} = \left\{ f_l \mid f \in \mathcal{F}, l \in S_f \right\}$ where for all $f \in \mathcal{F}$ and $l \in S_f$, $\mathtt{ar}(f_l) = \mathtt{ar}(f)$ holds. Furthermore, for every function symbol $f \in \mathcal{F}$, a map $\pi_f : M^n \mapsto S_f$ is chosen.

Definition 6.1.8 (Labeling Function, [Zan95])**.** Let $\mathcal{A}$ be an $\mathcal{F}$-algebra. Furthermore, for each $f \in \mathcal{F}$, let S_f be a non-empty set of labels and π_f a mapping as described above. The *labeling function* $\mathtt{lab} : \mathcal{T}(\mathcal{F}, X) \times A^X \mapsto \mathcal{T}(\overline{\mathcal{F}}, X)$ is a mapping from terms, together with a variable assignment, to labeled terms, defined inductively as follows:

$$\begin{array}{rcl} \mathtt{lab}(x, \alpha) & = & x \\ \mathtt{lab}(f(t_1, \ldots, t_n), \alpha) & = & f_{\pi_f([\alpha](t_1), \ldots, [\alpha](t_n))}(\mathtt{lab}(t_1, \alpha), \ldots, \mathtt{lab}(t_n, \alpha)) \end{array}$$

Example 6.1.9. Consider the signature $\mathcal{F}$ and the $\mathcal{F}$-algebra $\mathcal{A}$ from Example 6.1.2. Define the following sets of labels for the function symbols: $S_{\mathsf{0}} = \left\{0\right\}$, $S_{\mathsf{s}} = \left\{\mathbb{N}_0\right\}$ and $S_{\mathsf{plus}} = \left\{\mathbb{N}_0 \times \mathbb{N}_0\right\}$. Furthermore, let $\pi_{\mathsf{0}} = 0$, $\pi_{\mathsf{s}}(x) = x$ and $\pi_{\mathsf{plus}}(x, y) = (x, y)$. Let α be a variable assignment, where $\alpha(x) = 2$ and $\alpha(y) = 1$. The term $\mathsf{plus}(\mathsf{s}(\mathsf{x}),\mathsf{y})$ is labeled in the following way under the variable assignment α:

$$\mathtt{lab}(\mathsf{plus}(\mathsf{s}(\mathsf{x}),\mathsf{y}), \alpha) = \mathsf{plus}_{\pi_{\mathsf{plus}}(3,1)}(\mathsf{s}_{\pi_{\mathsf{s}}(2)}(x), y) = \mathsf{plus}_{(3,1)}(\mathsf{s}_{(2)}(x), y)$$

The labeling function can be used to label a TRS $\mathcal{R}$. To this end, a model for $\mathcal{R}$ together with the sets S_f and label mappings π_f are chosen. The definition below describes, how the labeled version of $\mathcal{R}$, denoted as $\overline{\mathcal{R}}$, is obtained from these parameters.

Definition 6.1.10 (Labeled TRS, [Zan95])**.** Let $\mathcal{R} = (\mathcal{F}, R)$ be a TRS, $\mathcal{M}$ a model for $\mathcal{R}$. The labeled TRS $\overline{\mathcal{R}} = (\overline{R}, \overline{\mathcal{F}})$ has the following rules:

$$\overline{R} = \left\{ \mathtt{lab}(l, \alpha) \to \mathtt{lab}(r, \alpha) \mid l \to r \in R, \alpha \in M^X \right\}$$

The main theorem of semantic labeling says that any TRS that is labeled in the way described above is terminating if and only if the unlabeled version is terminating.

Theorem 6.1.11 (Semantic Labeling, [Zan95]). *Let $\mathcal{M}$ be a model for a TRS $\mathcal{R}$ over a signature $\mathcal{F}$. Choose for every $f \in \mathcal{F}$ a non-empty set S_f of labels and a map $\pi_f : M^n \mapsto S_f$, where n is the arity of f. Define $\overline{\mathcal{R}}$ as above. Then $\mathcal{R}$ is terminating if and only if $\overline{\mathcal{R}}$ terminating.*

Example 6.1.12 ([Zan95]). Consider the following TRS $\mathcal{R}$ which is not simply terminating:

$$\mathsf{f}(\mathsf{f}(\mathsf{x})) \quad \to \quad \mathsf{f}(\mathsf{g}(\mathsf{f}(\mathsf{x})))$$

Since $\mathcal{R}$ is not simply terminating, no simplification order can orient this TRS. Now we apply semantic labeling using the following parameters:

- A model with the domain $M = \{1, 2\}$ and functions $f_{\mathcal{M}}(x) = 2$ and $g_{\mathcal{M}}(x) = 1$ for $x \in \{1, 2\}$.
- Labels for f and g: $S_f = \{1, 2\}$ and $S_g = \{1\}$.
- Label mappings: $\pi_f(x) = x$ and $\pi_g(x) = 1$ for $x \in \{1, 2\}$.

The labeled system $\overline{\mathcal{R}}$ is:

$$\mathsf{f}_2(\mathsf{f}_1(\mathsf{x})) \quad \to \quad \mathsf{f}_1(\mathsf{g}_1(\mathsf{f}_1(\mathsf{x}))) \qquad\qquad \mathsf{f}_2(\mathsf{f}_2(\mathsf{x})) \quad \to \quad \mathsf{f}_1(\mathsf{g}_1(\mathsf{f}_2(\mathsf{x})))$$

The system $\overline{\mathcal{R}}$ can easily be oriented using a KBO with a weight function (w, w_0) where $w_0 = 1$, $w(f_1) = w(g_1) = 1$ and $w(f_2) = 3$.

Semantic Labeling for Context-Sensitive Rewriting

Semantic labeling is easily generalized to context-sensitive rewriting. Almost all notions carry over from the non-context-sensitive case without changes. The only thing changed is, that when labeling a CS-TRS $(\mathcal{R}, \mu)$ with a labeling function, the replacement map μ needs to be extended to the new signature of $\overline{\mathcal{R}}$.

Definition 6.1.13 (Labeled CS-TRS). Let $(\mathcal{R}, \mu)$ be a CS-TRS over signature $\mathcal{F}$, $\mathcal{M}$ a model for $\mathcal{R}$. The labeled CS-TRS $(\overline{\mathcal{R}}, \overline{\mu})$ over signature $\overline{\mathcal{F}}$ has the following set of rules:

$$\overline{R} = \{\mathtt{lab}(l, \alpha) \to \mathtt{lab}(r, \alpha) \mid l \to r \in R, \alpha \in M^X\}$$

and for each $f_l \in \overline{\mathcal{F}}$, we define $\overline{\mu}(f_l) = \mu(f)$ (here f denotes the symbol f_l where the label is removed).

We state an axillary result from [Zan95] that will be helpful in the proofs, when extending semantic labeling to the context-sensitive case. The other results in this section are generalizations of the results from the same paper.

Lemma 6.1.14 ([Zan95])**.** *Let $\alpha \in A^X$ be a variable assignment and let σ be a substitution. Define $\bar{\sigma} : X \mapsto \mathcal{T}(\overline{\mathcal{F}}, X)$ by $\bar{\sigma}(x) = \mathtt{lab}(\sigma(x), \alpha)$. Then*

$$\mathtt{lab}(\sigma(t), \alpha) = \bar{\sigma}(\mathtt{lab}(t, [\alpha] \circ \sigma))$$

Lemma 6.1.15. *Let $\mathcal{M}$ be a model for a CS-TRS $(\mathcal{R}, \mu)$ and let s, t be terms such that $s \hookrightarrow_{\mathcal{R},\mu} t$. Then*

$$\mathtt{lab}(s, \alpha) \hookrightarrow_{\overline{\mathcal{R}},\overline{\mu}} \mathtt{lab}(t, \alpha)$$

for all variable assignments $\alpha \in M^X$.

Proof. By induction on the depth of the position, where the rewrite step is applied. Suppose, the reduction is applied at root position of s. Then there is a substitution σ such that $s = \sigma(l)$ and $t = \sigma(r)$ for some rule $l \to r \in \mathcal{R}$. By construction of $\overline{\mathcal{R}}$, we know that $\mathtt{lab}(l, [\alpha] \circ \sigma) \to \mathtt{lab}(r, [\alpha] \circ \sigma)$ is a rule in $\overline{\mathcal{R}}$. Applying Lemma 6.1.14 twice, we get:

$$\begin{array}{rcll} \mathtt{lab}(s, \alpha) & = & \mathtt{lab}(\sigma(l), \alpha) & \\ & = & \bar{\sigma}(\mathtt{lab}(l, [\alpha] \circ \sigma)) & \\ & \overset{\epsilon}{\hookrightarrow}_{\overline{\mathcal{P}},\overline{\mu}} & \bar{\sigma}(\mathtt{lab}(r, [\alpha] \circ \sigma)) & \\ & = & \mathtt{lab}(\sigma(r), \alpha) & = \quad \mathtt{lab}(t, \alpha) \end{array}$$

For the induction step, suppose $s = f(\dots, s_{i-1}, s', s_{i+1}, \dots)$ and $t = f(\dots, s_{i-1}, t', s_{i+1}, \dots)$ with $s \hookrightarrow_{\mathcal{R},\mu} t$. For this step to be possible, $i \in \mu(f)$ must hold. From the induction hypothesis we know, that there is the following rewrite step: $\mathtt{lab}(s', \alpha) \hookrightarrow_{\overline{\mathcal{P}},\overline{\mu}} \mathtt{lab}(t', \alpha)$. Furthermore, we know that $[\alpha](s') = [\alpha](t')$ because $\mathcal{A}$ is a model of $\mathcal{R}$. Thus we have (terms abbreviated):

$$\begin{array}{rcll} \mathtt{lab}(f(\dots, s', \dots), \alpha) & = & f_{\pi_f(\dots,[\alpha](s'),\dots)}(\dots, \mathtt{lab}(s', \alpha), \dots) & (1) \\ & = & f_{\pi_f(\dots,[\alpha](t'),\dots)}(\dots, \mathtt{lab}(s', \alpha), \dots) & (2) \\ & \hookrightarrow_{\overline{\mathcal{R}},\overline{\mu}} & f_{\pi_f(\dots,[\alpha](t'),\dots)}(\dots, \mathtt{lab}(t', \alpha), \dots) & (3) \\ & = & \mathtt{lab}(f(\dots, t', \dots), \alpha) & (4) \end{array}$$

The rewrite step used to get from line (2) to line (3) is valid, because by construction of $\overline{\mu}$ we have $\overline{\mu}(f_{\pi_f(\dots,[\alpha](t'),\dots)}) = \mu(f)$ and thus $i \in \overline{\mu}(f_{\pi_f(\dots,[\alpha](t'),\dots)})$. □

Now, we can formulate the theorem stating equivalence of CS-TRSs $\mathcal{R}$ and $\overline{\mathcal{R}}$ regarding termination. This is exactly analogous to Theorem 6.1.11:

Theorem 6.1.16. *Let $\mathcal{M}$ be a model for a CS-TRS $(\mathcal{R}, \mu)$ over $\mathcal{F}$. Choose for every $f \in \mathcal{F}$ a non-empty set S_f of labels and a map $\pi_f : M^n \mapsto S_f$, where n is the arity of f. Define $(\overline{\mathcal{R}}, \overline{\mu})$ as above. Then $\mathcal{R}$ is terminating if and only if $(\overline{\mathcal{R}}, \overline{\mu})$ terminating.*

Proof. If $(\overline{\mathcal{R}}, \overline{\mu})$ is not terminating, then there is an infinite rewrite sequence of terms in $\mathcal{T}(\overline{\mathcal{F}}, X)$. Removing all labels yields an infinite rewrite sequence in $\mathcal{R}$. For the other direction, assume that $\mathcal{R}$ is not terminating. Then there is an infinite rewrite sequence

$$t_1 \hookrightarrow_{\mathcal{R},\mu} t_2 \hookrightarrow_{\mathcal{R},\mu} t_3 \hookrightarrow_{\mathcal{R},\mu} \dots$$

Choose α arbitrarily. Then, according to Lemma 6.1.15, $\overline{\mathcal{R}}$ allows an infinite reduction

$$\mathtt{lab}(t_1, \alpha) \hookrightarrow_{\overline{\mathcal{R}},\overline{\mu}} \mathtt{lab}(t_2, \alpha) \hookrightarrow_{\overline{\mathcal{R}},\overline{\mu}} \mathtt{lab}(t_3, \alpha) \hookrightarrow_{\overline{\mathcal{R}},\overline{\mu}} \dots$$

for some variable assignment α. □

Example 6.1.17. Consider the TRS $\mathcal{R}$ and the labeling parameters from Example 6.1.12. Suppose, that instead of $\mathcal{R}$, we now have a context-sensitive TRS $(\mathcal{R}, \mu)$ with $\mu(f) = \{1, 2\}$ and $\mu(g) = \emptyset$. The labeled version of this CS-TRS is $(\overline{\mathcal{R}}, \overline{\mu})$, where $\overline{\mathcal{R}}$ is the same as in Example 6.1.12. For the replacement map $\overline{\mu}$ we have: $\overline{\mu}(f_1) = \overline{\mu}(f_2) = \{1, 2\}$ and $\overline{\mu}(g) = \emptyset$

Semantic Labeling as a CS-DP processor

Now, we discuss how semantic labeling can be used as a CS-DP processor. This extension is based on the DP processor for root labeling (see next section), described in [SM08]. However, here it is generalized in two ways: first, it supports context-sensitive rewriting and second, it supports arbitrary types of semantic labeling. We state a few helpful definitions from [SM08] regarding signatures of DP problems.

Definition 6.1.18. Let $(\mathcal{P}, \mathcal{R}, \mu)$ be a CS-DP problem where $\mathcal{P}$ is a TRS over signature $\mathcal{F}_\mathcal{P}$ and $\mathcal{R}$ is a TRS over signature $\mathcal{F}_\mathcal{R}$. Let the signature $\mathcal{F}^{(\#)}$ be defined as $\{\mathtt{root}(l), \mathtt{root}(r) \mid l \to r \in \mathcal{P}\}$ and define $\mathcal{F}$ as $\mathcal{F}_\mathcal{R} \cup (\mathcal{F}_\mathcal{P} \setminus \mathcal{F}^{(\#)})$. Furthermore, we define the signature $\mathcal{F}(\mathcal{R})$ to contain only the function symbols occuring in $\mathcal{R}$. Sometimes we augment the notation with position constraints, so for example $\mathcal{F}_{>\epsilon}(\mathcal{R})$ denotes the signature that contains only the function symbols occurring *below root* in $\mathcal{R}$.

Semantic labeling is applied to a CS-DP problem by labeling it in a way similar to how CS-TRSs are labeled:

Definition 6.1.19 (Labeled CS-DP Problem)**.** Let $(\mathcal{P}, \mathcal{R}, \mu)$ be a CS-DP problem and let $\mathcal{F}^{(\#)}$ and $\mathcal{F}$ be defined as above. Let $\mathcal{A}$ be a $(\mathcal{F} \cup \mathcal{F}^{(\#)})$-algebra. For each $f \in \mathcal{F} \cup \mathcal{F}^{(\#)}$, choose a set S_f and a mapping $\pi_f : A^n \mapsto S_f$, where n is the arity of f. By using Definition 6.1.13, we can label the CS-TRSs $(\mathcal{P}, \mu)$ and $(\mathcal{R}, \mu)$ and obtain $(\overline{\mathcal{P}}, \overline{\mu}_\mathcal{P})$ and $(\overline{\mathcal{R}}, \overline{\mu}_\mathcal{R})$. Let $\overline{\mu}(f) = \overline{\mu}_\mathcal{R}(f)$ for all f in $\mathcal{F}$ and let $\overline{\mu}(F) = \overline{\mu}_\mathcal{P}(F)$ for all $F \in \mathcal{F}^{(\#)}$. Now we can define the labeled version of the CS-DP problem $(\mathcal{P}, \mathcal{R}, \mu)$ as $(\overline{\mathcal{P}}, \overline{\mathcal{R}}, \overline{\mu})$.

Using these definitions, we are now ready to state the theorem defining the CS-DP processor for semantic labeling.

Theorem 6.1.20. *Let $(\mathcal{P}, \mathcal{R}, \mu)$ be a CS-DP problem and let $\mathcal{F}^{(\#)}$ and $\mathcal{F}$ be defined as above. Let $\mathcal{A}$ be a $(\mathcal{F} \cup \mathcal{F}^{(\#)})$-algebra. Choose for every $f \in \mathcal{F} \cup \mathcal{F}^{(\#)}$ a set S_f of labels and a map $\pi_f : A^n \mapsto S_f$, where n is the arity of f. If $\mathcal{A}$ is a model of $\mathcal{P}$ and $\mathcal{R}$, then the DP Processor*

$$(\mathcal{P}, \mathcal{R}, \mu) \mapsto \{(\overline{\mathcal{P}}, \overline{\mathcal{R}}, \overline{\mu})\}$$

is sound and complete.

Remark. One way of obtaining a model for $\mathcal{P}$ is to choose special functions for the root-symbols of the rules in $\mathcal{P}$: for each $F \in \mathcal{F}^{(\#)}$ with arity $n \geq 0$, choose $F_{\mathcal{A}}(x_1, \ldots, x_n) = a$ for some arbitrary but fixed $a \in A$ Such an algebra is trivially a model for $\mathcal{P}$, as the left-hand-sides and right-hand-sides of the rules in $\mathcal{P}$ all evaluate to the same symbol a.

Soundness. Suppose $(\mathcal{P}, \mathcal{R}, \mu)$ is not finite. Then there is an infinite minimal sequence in $(\mathcal{P}, \mathcal{R}, \mu)$:

$$s_1 \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} t_1 \hookrightarrow^*_{\mathcal{R},\mu} s_2 \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} t_2 \hookrightarrow^*_{\mathcal{R},\mu} \ldots$$

Let $\alpha \in A^X$ be an arbitrary variable assignment. Consider an arbitrary step $s_i \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} t_i$ $(i \geq 1)$ in the assumed sequence. There is a substitution σ such that for some rule $l \to r \in \mathcal{P}$, we have $s_i = \sigma(l)$ and $t_i = \sigma(r)$, because the rewrite step takes place at the root position of s_i. By using Lemma 6.1.13 twice we get

$$\begin{array}{rcll} \mathtt{lab}(s_i, \alpha) & = & \mathtt{lab}(\sigma(l), \alpha) & \\ & = & \bar{\sigma}(\mathtt{lab}(l, [\alpha] \circ \sigma)) & \\ & \overset{\epsilon}{\hookrightarrow}_{\overline{\mathcal{P}},\overline{\mu}} & \bar{\sigma}(\mathtt{lab}(r, [\alpha] \circ \sigma)) & \\ & = & \mathtt{lab}(\sigma(r), \alpha) & = \quad \mathtt{lab}(t_i, \alpha) \end{array}$$

because the rule $\mathtt{lab}(l, [\alpha] \circ \sigma) \to \mathtt{lab}(r, [\alpha] \circ \sigma)$ is in $\overline{\mathcal{P}}$. Since the rule is applied at root position, it is applicable regardless of $\overline{\mu}$.
Now, consider the sequence $t_i \hookrightarrow^*_{\mathcal{R},\mu} s_{i+1}$ $(i \geq 1)$ occurring in the assumed sequence. Let $s \hookrightarrow_{\mathcal{R},\mu} t$ be an arbitrary step within this sequence. By Lemma 6.1.15 we know, that there is the following rewrite step in $\overline{\mathcal{R}}$: $\mathtt{lab}(s, \alpha) \hookrightarrow_{\overline{\mathcal{R}},\overline{\mu}} \mathtt{lab}(t, \alpha)$ and by repeating this, we obtain $\mathtt{lab}(t_i, \alpha) \hookrightarrow^*_{\overline{\mathcal{R}},\overline{\mu}} \mathtt{lab}(s_{i+1}, \alpha)$
Therefore, we have constructed an infinite sequence in $(\overline{\mathcal{P}}, \overline{\mathcal{R}}, \overline{\mu})$:

$$\mathtt{lab}(s_1, \alpha) \overset{\epsilon}{\hookrightarrow}_{\overline{\mathcal{P}},\overline{\mu}} \mathtt{lab}(t_1, \alpha) \hookrightarrow^*_{\overline{\mathcal{R}},\overline{\mu}} \mathtt{lab}(s_2, \alpha) \overset{\epsilon}{\hookrightarrow}_{\overline{\mathcal{P}},\overline{\mu}} \mathtt{lab}(t_2, \alpha) \hookrightarrow^*_{\overline{\mathcal{R}},\overline{\mu}} \ldots$$

Now, we argue that this sequence is also minimal (i.e. every term $\mathtt{lab}(t_i, \alpha)$ is terminating with respect to $(\overline{\mathcal{R}}, \overline{\mu})$). Suppose some term $\mathtt{lab}(t_i, \alpha)$ is not terminating with respect to $(\overline{\mathcal{R}}, \overline{\mu})$. Then, there is an infinite rewrite sequence starting at this term. By removing all labels we get an infinite rewrite sequence in $(\mathcal{R}, \mu)$ starting at t_i which is a contradiction to minimality of the assumed sequence. □

Completeness. Suppose, $(\overline{\mathcal{P}}, \overline{\mathcal{R}}, \overline{\mu})$ is not finite. Then there is an infinite minimal sequence in $(\overline{\mathcal{P}}, \overline{\mathcal{R}}, \overline{\mu})$ for some $\alpha \in A^X$:

$$\mathtt{lab}(s_1, \alpha) \overset{\epsilon}{\hookrightarrow}_{\overline{\mathcal{P}},\overline{\mu}} \mathtt{lab}(t_1, \alpha) \hookrightarrow^*_{\overline{\mathcal{R}},\overline{\mu}} \mathtt{lab}(s_2, \alpha) \overset{\epsilon}{\hookrightarrow}_{\overline{\mathcal{P}},\overline{\mu}} \mathtt{lab}(t_2, \alpha) \hookrightarrow^*_{\overline{\mathcal{R}},\overline{\mu}} \ldots$$

Removing all labels in this sequence yields the following infinite sequence in $(\mathcal{P}, \mathcal{R}, \mu)$:

$$s_1 \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} t_1 \hookrightarrow^*_{\mathcal{R},\mu} s_2 \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} t_2 \hookrightarrow^*_{\mathcal{R},\mu} \ldots$$

This sequence is also minimal: Suppose, some term t_i is not terminating with respect to $(\mathcal{R}, \mu)$. Then there is an infinite rewrite sequence $t_i = u_1 \hookrightarrow_{\mathcal{R},\mu} u_2 \hookrightarrow_{\mathcal{R},\mu} \ldots$. According to Lemma 6.1.15, since $\mathcal{A}$ is a model of $\mathcal{R}$, there is an infinite sequence $\mathtt{lab}(t_i, \alpha) = \mathtt{lab}(u_1, \alpha) \hookrightarrow_{\overline{\mathcal{R}},\overline{\mu}} \mathtt{lab}(u_2, \alpha) \hookrightarrow_{\overline{\mathcal{R}},\overline{\mu}} \ldots$ which contradicts minimality of the assumed infinite minimal sequence. □

6.2 Root-Labeling

In order to use semantic labeling, many parameters have to be fixed: A model for the TRS $\mathcal{R}$ that should be proven terminating, and label sets as well as label mappings for each function symbol in the signature. Also, if the label sets are not finite, the resulting TRS $\overline{\mathcal{R}}$ might be infinite. This huge search space makes automated proving of termination with unrestricted semantic labeling quite hard. Therefore, techniques like self-labelling [MOZ96] and root-labeling [SM08] have been developed, where the labeling of a TRS $\mathcal{R}$ is determined solely by $\mathcal{R}$ itself.

One part of this thesis is the implementation of root-labeling as a direct method and as a CS-DP processor for VMTL. Root-labeling was originally developed for string-rewriting systems (a special case of term rewriting, where the signature is restricted to unary symbols) by Johannes Waldmann and generalized to term rewriting by Christian Sternagel and Aart Middeldorp in [SM08]. In this section, we recapitulate some of the definitions and results for root-labeling from [SM08]. In the next section, we generalize these results to the context-sensitive case for the direct method and then for the DP processor.

The following definitions will be needed in the subsequent sections.

Definition 6.2.1 (Root Preserving / Altering Rules, [SM08])**.** Let $\mathcal{R}$ be a TRS. The rules in the set $\mathcal{R}_p = \{l \to r \in \mathcal{R} \mid \mathtt{root}(l) = \mathtt{root}(r)\}$ are called *root preserving*. The rules in the set $\mathcal{R}_a = \mathcal{R} \setminus \mathcal{R}_p$ are called *root altering*.

Definition 6.2.2 (Flat Contexts, [SM08])**.** Let $\mathcal{R}$ be a TRS over signature $\mathcal{F}$. The set $\mathcal{FC}_\mathcal{F} = \{f(x_1, \ldots, x_{i-1}, \Box, x_{i+1}, \ldots, x_n) \mid f \in \mathcal{F}^{(n)}, 1 \leq i \leq n, n \geq 1\}$ is the set of *flat contexts*. The *flat context closure* of $\mathcal{R}$ under a signature $\mathcal{F}$ is defined as $\mathcal{FC}_\mathcal{F}(\mathcal{R}) = \mathcal{R}_p \cup \{C[l] \to C[r] \mid l \to r \in \mathcal{R}_a, C \in \mathcal{FC}_\mathcal{F}\}$.

The Direct Method

The root-labeling transformation of a system $\mathcal{R}$ is a special case of the semantic labeling transformation where as an algebra for $\mathcal{R}$, one uses the algebra $\mathcal{A}_\mathcal{F}$ with domain $\mathcal{F}$ and functions mapping to the corresponding function symbols from $\mathcal{F}$.

Definition 6.2.3 (The Algebra $\mathcal{A}_\mathcal{F}$, [SM08])**.** Let $\mathcal{F}$ be a signature. The algebra $\mathcal{A}_\mathcal{F}$ is defined as $(\mathcal{F}, \{f_{\mathcal{A}_\mathcal{F}} \mid f \in \mathcal{F}\})$ where for all function symbols in $\mathcal{F}$ with arity $n \geq 0$, we define $f_{\mathcal{A}_\mathcal{F}}(x_1, \ldots, x_n) = f$ for all $x_1, \ldots, x_n \in \mathcal{F}$.

The next definition describes, how the labels for root-labeled terms are chosen. The whole translation is determined by the TRS and its signature, and thus does not require any parameters, making it very well suited for automation. However, the proofs found in this way are not

very suited for understandability by humans, because the labeling does not reflect the intended, natural semantics of the system. Thus some of the elegance of the semantic labeling approach is sacrificed for the improved efficiency.

Definition 6.2.4 (Root Labeling, [SM08])**.** Let $\mathcal{R}$ be a TRS over signature $\mathcal{F}$. Let $\mathcal{A}_\mathcal{F}$ be as defined above. Choose for every $f \in \mathcal{F}^{(n)}$ with $n \geq 1$ the set of labels $S_f = \mathcal{F}^n$ and mapping $\pi_f(x_1, \ldots, x_n) = (x_1, \ldots, x_n)$ for all $x_1, \ldots, x_n \in \mathcal{F}$. For each $c \in \mathcal{F}^{(0)}$, choose $S_c = \{c\}$ and $\pi_c = c$. A system labeled by semantic labeling with these parameters is called *root-labeled*, and is denoted as $\mathcal{R}_{\mathtt{rl}}$.

Example 6.2.5. Consider the TRS $\mathcal{R}$ (from [Toy87]) consisting only of the rule $\mathsf{f}(\mathsf{a}, \mathsf{b}, x) \to \mathsf{f}(x, x, x)$. The root-labeled version, $\mathcal{R}_{\mathtt{rl}}$, is:

$$\begin{array}{lcllcl} \mathsf{f}_{(\mathsf{a},\mathsf{b},\mathsf{a})}(\mathsf{a}, \mathsf{b}, x) & \to & \mathsf{f}_{(\mathsf{a},\mathsf{a},\mathsf{a})}(x, x, x) & \mathsf{f}_{(\mathsf{a},\mathsf{b},\mathsf{b})}(\mathsf{a}, \mathsf{b}, x) & \to & \mathsf{f}_{(\mathsf{b},\mathsf{b},\mathsf{b})}(x, x, x) \\ \mathsf{f}_{(\mathsf{a},\mathsf{b},\mathsf{f})}(\mathsf{a}, \mathsf{b}, x) & \to & \mathsf{f}_{(\mathsf{f},\mathsf{f},\mathsf{f})}(x, x, x) & & & \end{array}$$

While $\mathcal{R}$ is not simply-terminating, and thus any simplification-order like RPO and KBO would fail to prove termination of $\mathcal{R}$, the root-labeled system $\mathcal{R}_{\mathtt{rl}}$ is simply terminating and termination can be proved very easily using the RPO induced by a strict order $>$ s.t. $\mathsf{f}_{(\mathsf{a},\mathsf{b},\mathsf{a})} > \mathsf{f}_{(\mathsf{a},\mathsf{a},\mathsf{a})}$, $\mathsf{f}_{(\mathsf{a},\mathsf{b},\mathsf{b})} > \mathsf{f}_{(\mathsf{b},\mathsf{b},\mathsf{b})}$ and $\mathsf{f}_{(\mathsf{a},\mathsf{b},\mathsf{f})} > \mathsf{f}_{(\mathsf{f},\mathsf{f},\mathsf{f})}$.

Definition 6.2.6 (Flat-Context Stability)**.** We say, a TRS $\mathcal{R}$ over a signature $\mathcal{F}$ is *flat-context stable*, if whenever there are root-altering rules in $\mathcal{R}$, then there is at least one function symbol $f \in \mathcal{F}$ with arity $\mathtt{ar}(f) \geq 1$.

Theorem 6.2.7 (Termination of Root-Labeled Systems, [SM08])**.** *Let $\mathcal{R}$ be a flat-context stable TRS over signature $\mathcal{F}$. Then the root-labeled system $\mathcal{FC}_\mathcal{F}(\mathcal{R})_{\mathtt{rl}}$ is terminating iff $\mathcal{R}$ is terminating.*

Remark. Note, that the flat context closure is indeed required, because $\mathcal{A}_\mathcal{F}$ is not necessarily a model for $\mathcal{R}$, but by construction is a model of $\mathcal{FC}_\mathcal{F}(\mathcal{R})$ [SM08]. Furthermore, flat context-stability is required, because otherwise the set of flat-contexts would be empty.

The DP Processor

The extension of root-labeling to a DP processor was first presented in [SM08]. Like the direct method version, it is based on the flat context closure of the rule system. However, in [ST10] it was shown that in general the restriction to the implicit signature of the DP problem, which is inherent to using flat contexts, is not sound if one needs to maintain minimality of DP chains [SM08, Lemma 13]. It is shown, however, that root-labeling is sound for DP problems where the set of rules is left-linear. Since in our setting minimality of chains is required, we will need this restriction to left-linear systems.

Definition 6.2.8 (Flat Context Closure of DP Problems, [SM08])**.** Let $(\mathcal{P}, \mathcal{R})$ be a DP problem, and let $\mathcal{F}^{(\#)}$ and $\mathcal{F}$ be as in Definition 6.1.18. Let Δ be a fresh unary function symbol. The function $\mathtt{block}$ inserts Δ between the root symbol f and the arguments $t_1, \ldots, t_n$ of a term $f(t_1, \ldots, t_n)$ with $n \geq 1$:

$$\mathtt{block}(t) = \begin{cases} t & \text{if } t \text{ is a variable or a constant} \\ f(\Delta(t_1), \ldots, f(\Delta(t_n)) & \text{if } t = f(t_1, \ldots t_n) \text{ for terms } t_1, \ldots t_n \end{cases}$$

Define $\mathcal{FC}(\mathcal{P}, \mathcal{R})$ as $(\mathtt{block}(\mathcal{P}), \mathcal{FC}_{\mathcal{F} \cup \{\Delta\}}(\mathcal{R}))$ where $\mathtt{block}(\mathcal{P})$ is the TRS with the set of rules $\{\mathtt{block}(l) \to \mathtt{block}(r) \mid l \to r \in \mathcal{P}\}$.

Definition 6.2.9 (Root-Labeling Transformation for DP Problems, [SM08])**.** Let $(\mathcal{P}, \mathcal{R})$ be a DP problem and let $\mathcal{F}^{(\#)}$ and $\mathcal{F}$ be as in Definition 6.1.18. Let $\mathcal{F}' = \mathcal{F} \cup \mathcal{F}^{(\#)} \cup \{\Delta\}$. Let $\mathcal{P}'$ be the TRS with the rules from $\mathtt{block}(\mathcal{P})$, over the signature $\mathcal{F}'$ and let $\mathcal{R}'$ be the TRS with rules from $\mathcal{FC}_{\mathcal{F} \cup \{\Delta\}}(\mathcal{R})$ over signature $\mathcal{F}'$. The root-labeling transformation $\mathcal{FC}(\mathcal{P}, \mathcal{R})_{\mathtt{rl}}$ is defined as the pair $(\mathcal{P}'_{\mathtt{rl}}, \mathcal{R}'_{\mathtt{rl}})$, where the root-labeling operation is modified in the following way: for each function symbol F in $\mathcal{F}^{(\#)}$, the set of labels is $S_F = \{F\}$, the label mapping is $\pi_F = F$ and the interpretation of F in $\mathcal{A}_{\mathcal{F}'}$ is $F_{\mathcal{A}_{\mathcal{F}'}}(x_1, \ldots, x_n) = g$ for all $x_1, \ldots, x_n \in \mathcal{F}'$ and arbitrary but fixed $g \in \mathcal{F}^{(\#)}$.

Example 6.2.10. Consider again the TRS $\mathcal{R}$ from Example 6.2.4. The corresponding DP problem $(DP(\mathcal{R}), \mathcal{R})$ contains the following dependency pair: $\mathsf{f}^{\#}(\mathsf{a}, \mathsf{b}, x) \to \mathsf{f}^{\#}(x, x, x)$. The root-labeled version, $\mathcal{FC}(DP(\mathcal{R}), \mathcal{R})_{\mathtt{rl}}$, contains the following dependency pairs and rules:

Dependency Pairs:

$$\begin{array}{rcl}
{\mathsf{f}_{(\Delta,\Delta,\Delta)}}^{\#}(\Delta_{\mathsf{a}}(\mathsf{a}), \Delta_{\mathsf{b}}(\mathsf{b}), \Delta_{\mathsf{a}}(x)) & \to & {\mathsf{f}_{(\Delta,\Delta,\Delta)}}^{\#}(\Delta_{\mathsf{a}}(x), \Delta_{\mathsf{a}}(x), \Delta_{\mathsf{a}}(x)) \\
{\mathsf{f}_{(\Delta,\Delta,\Delta)}}^{\#}(\Delta_{\mathsf{a}}(\mathsf{a}), \Delta_{\mathsf{b}}(\mathsf{b}), \Delta_{\mathsf{b}}(x)) & \to & {\mathsf{f}_{(\Delta,\Delta,\Delta)}}^{\#}(\Delta_{\mathsf{b}}(x), \Delta_{\mathsf{b}}(x), \Delta_{\mathsf{b}}(x)) \\
{\mathsf{f}_{(\Delta,\Delta,\Delta)}}^{\#}(\Delta_{\mathsf{a}}(\mathsf{a}), \Delta_{\mathsf{b}}(\mathsf{b}), \Delta_{\mathsf{f}}(x)) & \to & {\mathsf{f}_{(\Delta,\Delta,\Delta)}}^{\#}(\Delta_{\mathsf{f}}(x), \Delta_{\mathsf{f}}(x), \Delta_{\mathsf{f}}(x)) \\
{\mathsf{f}_{(\Delta,\Delta,\Delta)}}^{\#}(\Delta_{\mathsf{a}}(\mathsf{a}), \Delta_{\mathsf{b}}(\mathsf{b}), \Delta_{\mathsf{f}^{\#}}(x)) & \to & {\mathsf{f}_{(\Delta,\Delta,\Delta)}}^{\#}(\Delta_{\mathsf{f}^{\#}}(x), \Delta_{\mathsf{f}^{\#}}(x), \Delta_{\mathsf{f}^{\#}}(x)) \\
{\mathsf{f}_{(\Delta,\Delta,\Delta)}}^{\#}(\Delta_{\mathsf{a}}(\mathsf{a}), \Delta_{\mathsf{b}}(\mathsf{b}), \Delta_{\Delta}(x)) & \to & {\mathsf{f}_{(\Delta,\Delta,\Delta)}}^{\#}(\Delta_{\Delta}(x), \Delta_{\Delta}(x), \Delta_{\Delta}(x))
\end{array}$$

Rules:

$$\begin{array}{rcl}
\mathsf{f}_{(\mathsf{a},\mathsf{b},\mathsf{a})}(\mathsf{a}, \mathsf{b}, x) & \to & \mathsf{f}_{(\mathsf{a},\mathsf{a},\mathsf{a})}(x, x, x) \\
\mathsf{f}_{(\mathsf{a},\mathsf{b},\mathsf{f})}(\mathsf{a}, \mathsf{b}, x) & \to & \mathsf{f}_{(\mathsf{f},\mathsf{f},\mathsf{f})}(x, x, x) \\
\mathsf{f}_{(\mathsf{a},\mathsf{b},\mathsf{f}^{\#})}(\mathsf{a}, \mathsf{b}, x) & \to & \mathsf{f}_{(\mathsf{f}^{\#},\mathsf{f}^{\#},\mathsf{f}^{\#})}(x, x, x)
\end{array}$$

$$\begin{array}{rcl}
\mathsf{f}_{(\mathsf{a},\mathsf{b},\mathsf{b})}(\mathsf{a}, \mathsf{b}, x) & \to & \mathsf{f}_{(\mathsf{b},\mathsf{b},\mathsf{b})}(x, x, x) \\
\mathsf{f}_{(\mathsf{a},\mathsf{b},\Delta)}(\mathsf{a}, \mathsf{b}, x) & \to & \mathsf{f}_{(\Delta,\Delta,\Delta)}(x, x, x)
\end{array}$$

Theorem 6.2.11 (The Root-Labeling DP Processor [SM08, ST10])**.** *Let $(\mathcal{P}, \mathcal{R})$ be a DP problem. The following DP processor is sound:*

$$(\mathcal{P}, \mathcal{R}) \mapsto \begin{cases} \{\mathcal{FC}(\mathcal{P}, \mathcal{R})_{\mathtt{rl}}\} & \textit{if } \mathcal{R} \textit{ is left-linear} \\ \{(\mathcal{P}, \mathcal{R})\} & \textit{otw.} \end{cases}$$

6.3 Generalizing Root-Labeling to the Context-Sensitive Case

In this section, we generalize root-labeling to the context-sensitive case. We start by generalizing the notion of flat contexts.

Definition 6.3.1 (Flat μ-Contexts)**.** Let $(\mathcal{R}, \mu)$ be a CS-TRS over the signature $\mathcal{F}$. Let $\mathcal{R}_a$ and $\mathcal{R}_p$ be as in Definition 6.1.18. The set of flat μ-contexts for a signature $\mathcal{F}$ is

$$\mathcal{FC}_{\mathcal{F},\mu} = \left\{ f(x_1, \ldots, x_{i-1}, \Box, x_{i+1}, \ldots, x_n) \mid f \in \mathcal{F}^{(n)}, n \geq 1, i \in \mu(f) \right\}$$

The flat μ-context closure of a CS-TRS $(\mathcal{R}, \mu)$ is defined as follows: $\mathcal{FC}_{\mathcal{F}}(\mathcal{R}, \mu) = \mathcal{R}_p \cup \left\{ C[l] \to C[r] \mid l \to r \in \mathcal{R}_a, C \in \mathcal{FC}_{\mathcal{F},\mu} \right\}$

Definition 6.3.2 (Flat μ-Context Stability)**.** We say, a CS-TRS $(\mathcal{R}, \mu)$ over a signature $\mathcal{F}$ is *flat μ-context stable*, if whenever there are root-altering rules in $\mathcal{R}$, then there is at least one function symbol $f \in \mathcal{F}$ with arity $\mathtt{ar}(f) \geq 1$ and $\mu(f) \neq \emptyset$.

Lemma 6.3.3. *Let $(\mathcal{R}, \mu)$ be a flat μ-context stable CS-TRS over the signature $\mathcal{F}$. The CS-TRS $(\mathcal{R}, \mu)$ is terminating if and only if the CS-TRS $(\mathcal{FC}_{\mathcal{F}}(\mathcal{R}, \mu), \mu)$ is terminating.*

Proof. Suppose, $(\mathcal{FC}_{\mathcal{F}}(\mathcal{R}, \mu), \mu)$ is not terminating. Then there is an infinite sequence

$$t_1 \hookrightarrow_{\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu),\mu} t_2 \hookrightarrow_{\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu),\mu} \cdots$$

We argue, that each step in this sequence can be simulated via steps in $\mathcal{R}$. Consider an arbitrary step $t_i \hookrightarrow_{\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu),\mu} t_{i+1}$. Let $l \to r \in \mathcal{FC}_{\mathcal{F}}(\mathcal{R}, \mu)$ be the rule applied at position π on t_i in order to obtain t_{i+1}. There are two possibilities:

- $l \to r \in \mathcal{R}_p$: Then also $l \to r \in \mathcal{R}$ by construction of $\mathcal{FC}_{\mu}(\mathcal{R})$.
- $l \to r \notin \mathcal{R}_p$: Then the rule $l \to r$ has the following shape: $C[l'] \to C[r']$ for some context $C = f(x_1, \ldots, x_{j-1}, \Box, x_{j+1}, \ldots, x_n) \in \mathcal{FC}_{\mathcal{F},\mu}$. By construction of $\mathcal{FC}_{\mathcal{F}}(\mathcal{R}, \mu)$ we have $j \in \mu(f)$ and $l' \to r' \in \mathcal{R}$. Furthermore, since the rewrite step took place at position π we know that π is a μ-replacing position in term t_i. Thus also $\pi.j$ is a μ-replacing position in t_i. Therefore, it is possible, to apply the rule $l' \to r'$ at position $\pi.j$ on t_i and obtain t_{i+1}.

Since every step in the infinite sequence $t_1 \hookrightarrow_{\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu),\mu} t_2 \hookrightarrow_{\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu),\mu} \ldots$ can be simulated with a rule from $\mathcal{R}$, we obtain the infinite sequence $t_1 \hookrightarrow_{\mathcal{R},\mu} t_2 \hookrightarrow_{\mathcal{R},\mu} \ldots$ and therefore, we know that $\mathcal{R}$ is not terminating.

For the other direction, assume $\mathcal{R}$ is non-terminating. In this case, there is an infinite rewrite sequence

$$t_1 \hookrightarrow_{\mathcal{R},\mu} t_2 \hookrightarrow_{\mathcal{R},\mu} \cdots$$

Let $C = f(x_1, \ldots, x_{j-1}, \Box, x_{j+1}, \ldots, x_n)$ be an arbitrary flat context from $\mathcal{FC}_{\mathcal{F},\mu}$. Then we know, $j \in \mu(f)$. Therefore, the following reduction sequence is also valid:

$$C[t_1] \hookrightarrow_{\mathcal{R},\mu} C[t_2] \hookrightarrow_{\mathcal{R},\mu} \ldots$$

The same sequence can be obtained using rules from $\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu)$ instead:

$$C[t_1] \hookrightarrow_{\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu),\mu} C[t_2] \hookrightarrow_{\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu),\mu} \cdots$$

This sequence can be constructed as follows: Consider again an arbitrary step $C[t_i] \hookrightarrow_{\mathcal{R},\mu} C[t_{i+1}]$. Let $l \to r \in \mathcal{R}$ be the rule that is applied at position π on the term $C[t_i]$, in order to obtain $C[t_{i+1}]$. Again, we consider the two cases:

- $l \to r \in \mathcal{R}_p$: Then $l \to r$ is also in $\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu)$ and this rule can be used.
- $l \to r \notin \mathcal{R}_p$: Note, that $C[t_i] = f(x_1,\ldots,x_{j-1},t_i,x_{j+1},\ldots,x_n)$. We distinguish two more cases:
  - $\pi = j$: (i.e. the rule is applied directly at the root position of the term t_i in its enclosing context). Then the rule $C[l] \to C[r] \in \mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu)$ can be applied at the root position of $C[t_i]$ to obtain $C[t_{i+1}]$.
  - $\pi > j$: Split π in two parts: $\pi = \pi'.k$ where π' is a position and $k \in \mathbb{N}$. Since π is a μ-replacing position in the term $C[t_i]$, the smaller position π' has to be μ-replacing as well. Let g denote the root symbol of the term $C[t_i]|_{\pi'}$. Because $\pi'.k$ is a μ-replacing position in $C[t_i]$, we also know, that $k \in \mu(g)$. Thus we know, that the rule $g(y_1,\ldots,y_{k-1},l,y_{k+1},\ldots,y_m) \to g(y_1,\ldots,y_{k-1},r,y_{k+1},\ldots,y_m)$ is in $\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu)$ and can be used at position π' on $C[t_i]$ to obtain $C[t_{i+1}]$.

Thus if $\mathcal{R}$ is not terminating, we can construct an infinite sequence using only rules from $\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu)$ and so this system is not terminating either. □

Now, we describe how a CS-TRS can be root-labeled. The labeling itself is identical to the non-context-sensitive case, but the replacement map has to be modified accordingly.

Definition 6.3.4 (Root-Labeling of CS-TRSs)**.** Let $(\mathcal{R},\mu)$ be a CS-TRS over the signature $\mathcal{F}$. Let the algebra $\mathcal{A}_{\mathcal{R}}$ be as in Definition 6.2.3. Choose for every $f \in \mathcal{F}^{(n)}$ with $n \geq 1$ the set of labels $S_f = \mathcal{F}^n$ and mapping $\pi_f(x_1,\ldots,x_n) = (x_1,\ldots,x_n)$ for all $x_i \in \mathcal{F}$. For each $c \in \mathcal{F}^{(0)}$, choose $S_c = \{c\}$ and $\pi_c = c$. A CS-TRS $(\mathcal{R},\mu)$ labeled by semantic labeling with these parameters is called *root-labeled*, and is denoted as $(\mathcal{R}_{\mathtt{rl}},\mu_{\mathtt{rl}})$ where $\mu_{\mathtt{rl}}(f_l) = \mu(f)$ for all f_l in the signature $\overline{\mathcal{F}}$ (cf. Section 6.1). f denotes the function symbol f_l where the label is removed.

Now, we can state the theorem expressing equivalence of termination of CS-TRSs and root-labeled CS-TRSs.

Theorem 6.3.5. *Let $(\mathcal{R},\mu)$ be a flat μ-context stable CS-TRS over the signature $\mathcal{F}$. The CS-TRS $(\mathcal{R},\mu)$ is terminating if and only if $(\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu)_{\mathtt{rl}},\mu_{\mathtt{rl}})$ is terminating.*

Proof. By the previous Lemma we know, that $(\mathcal{R},\mu)$ is terminating iff $(\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu),\mu)$ is terminating. Furthermore, we know that the algebra $\mathcal{A}_{\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu)}$ is a model for $\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu)$, because all rules in this system are root-preserving and thus the left-hand side and right-hand side of every rule trivially evaluate to the same element in $\mathcal{A}_{\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu)}$. Therefore, this theorem is a special case of Theorem 6.1.16. □

6.4 Generalizing the Root-Labeling Processor to the Context-Sensitive Case

In this section, we generalize the root-labeling processor to a context-sensitive DP processor.

Definition 6.4.1 (Flat Context Closure of CS-DP Problems)**.** Let $(\mathcal{P}, \mathcal{R}, \mu)$ be a CS-DP problem. Let the signatures $\mathcal{F}_\mathcal{P}$, $\mathcal{F}_\mathcal{R}$, $\mathcal{F}^{(\#)}$ and $\mathcal{F}$ be as in Definition 6.1.18 and let Δ and $\mathtt{block}$ be as in Definition 6.2.8. Define $\mathcal{FC}_\mathcal{F}(\mathcal{P}, \mathcal{R}, \mu)$ as the triple $(\mathtt{block}(\mathcal{P}), \mathcal{FC}_{\mathcal{F}\cup\{\Delta\}}(\mathcal{R}, \mu_\Delta), \mu_\Delta)$ where $\mathtt{block}(\mathcal{P}) = \{\mathtt{block}(l) \to \mathtt{block}(r) \mid l \to r \in \mathcal{P}\}$, and $\mu_\Delta(f) = \mu(f)$ for $f \in \mathcal{F} \cup \mathcal{F}^{(\#)}$ and $\mu_\Delta(\Delta) = \{1\}$.

Lemma 6.4.2. *The triple $\mathcal{FC}_\mathcal{F}(\mathcal{P}, \mathcal{R}, \mu)$ is a CS-DP problem.*

Proof. The only thing we need to show, is that the root symbols of rules in $\mathcal{P}$ occur only at root position in these rules and nowhere else. Since neither the $\mathtt{block}$-function nor closure under flat contexts introduce symbols from $\mathcal{F}^{(\#)}$, this is certainly the case if $(\mathcal{P}, \mathcal{R}, \mu)$ is a CS-DP problem (a prerequisite for applying the $\mathcal{FC}_\mathcal{F}$-operation). □

Before we can prove, that the flat context closure of a DP problem is finite if and only if the DP problem is finite, we need some more axillary results. First, we need a way to restrict terms over an arbitrary signature to some smaller signature. In [ST10] this is done by a *cleaning operation* which replaces every subterm of a term that is rooted by a symbol not in the smaller signature by some fixed variable:

Definition 6.4.3 (Cleaning [ST10])**.** Let t be a term over an arbitrary signature and let $\mathcal{F}$ be a signature. The cleaning operations removes from t (top-down) all occurrences of function symbols not occurring in $\mathcal{F}$ by replacing them with an arbitrary but fixed variable z:

$$[\![t]\!]_\mathcal{F} = \begin{cases} x & \text{if } t = x \in X \\ z & \text{if } t = f(t_1 \ldots, t_n) \text{ and } f \notin \mathcal{F} \\ f([\![t_1]\!]_\mathcal{F}, \ldots, [\![t_n]\!]_\mathcal{F}) & \text{if } t = f(t_1 \ldots, t_n) \text{ and } f \in \mathcal{F} \end{cases}$$

Cleaning can also be used on substitutions. Let σ be a substitution. For all $x \in X$ we define:

$$[\![\sigma]\!]_\mathcal{F}(x) = \begin{cases} z & \text{if } \sigma(x) = f(\ldots) \text{ and } f \notin \mathcal{F} \\ \sigma(x) & \text{otherwise} \end{cases}$$

Now, we show that every rewrite step from a term s to a term t over an arbitrary signature is also possible for cleaned versions of the terms or the cleaned versions of the terms are equal. The consequence of this result is that every reduction sequence from some term to some other term over an arbitrary signature is also possible for the cleaned versions.

Lemma 6.4.4. *Let $(\mathcal{R}, \mu)$ be a CS-TRS and let s and t be terms over an arbitrary signature such that $s \hookrightarrow_{\mathcal{R},\mu} t$ holds. Further, let $\mathcal{F}$ be some signature extending the implicit signature of $\mathcal{R}$ (i.e. $\mathcal{F} \supseteq \mathcal{F}(\mathcal{R})$). Then either $[\![s]\!]_\mathcal{F} \hookrightarrow_{\mathcal{R},\mu} [\![t]\!]_\mathcal{F}$ holds or $[\![s]\!]_\mathcal{F} = [\![t]\!]_\mathcal{F}$ holds.*

Proof. Suppose, that $s \hookrightarrow_{\mathcal{R},\mu} t$ holds by applying the rule $l \to r \in \mathcal{R}$ at position π in s to obtain t. We distinguish between two cases:

- Suppose, there is no position π' with $\epsilon \leq \pi' \leq \pi$ such that $\mathtt{root}(s|_{\pi'}) \notin \mathcal{F}$. We claim, that in this case the reduction step $[\![s]\!]_{\mathcal{F}} \hookrightarrow_{\mathcal{R},\mu} [\![t]\!]_{\mathcal{F}}$ is valid. We have $s = u[\sigma(l)]_\pi$ and $t = u[\sigma(r)]_\pi$ for some context u and substitution σ. From $s = u[\sigma(l)]_\pi$ we obtain $[\![s]\!]_{\mathcal{F}} = [\![u[\sigma(l)]_\pi]\!]_{\mathcal{F}}$. Since symbols that are not contained in $\mathcal{F}$ are not contained at or above π in this term, this is equivalent to $[\![u]\!]_{\mathcal{F}}[[\![\sigma(l)]\!]_{\mathcal{F}}]_\pi$. Furthermore, since l does not contain any symbol that is not in $\mathcal{F}$, we know that $[\![\sigma(l)]\!]_{\mathcal{F}}$ is equivalent to $[\![\sigma]\!]_{\mathcal{F}}(l)$ (because symbols not in $\mathcal{F}$ occurring in $\sigma(l)$ are necessarily introduced by the substitution σ, so replacing σ by $[\![\sigma]\!]_{\mathcal{F}}$ we directly substitute z then, thus simulating the effect of cleaning). So now we have $[\![s]\!]_{\mathcal{F}} = [\![u]\!]_{\mathcal{F}}[[\![\sigma]\!]_{\mathcal{F}}(l)]_\pi$. Using the same argument on $t = u[\sigma(r)]_\pi$, yields $[\![t]\!]_{\mathcal{F}} = [\![u]\!]_{\mathcal{F}}[[\![\sigma]\!]_{\mathcal{F}}(r)]_\pi$. Therefore, we can conclude that $[\![s]\!]_{\mathcal{F}} \hookrightarrow_{\mathcal{R},\mu} [\![t]\!]_{\mathcal{F}}$ is valid, by using the rule $l \to r$ instantiated by $[\![\sigma]\!]_{\mathcal{F}}$ at position π on the term $[\![s]\!]_{\mathcal{F}}$.

- For the complementary case, suppose that there is some position π' with $\epsilon \leq \pi' \leq \pi$ and $\mathtt{root}(s|_{\pi'}) \notin \mathcal{F}$. Note, that $\pi' \neq \pi$, because $\mathcal{R}$ does not contain any symbol not in $\mathcal{F}$. For this case, the claim is, that $[\![s]\!]_{\mathcal{F}}$ and $[\![t]\!]_{\mathcal{F}}$ are equal. To show this, consider the minimal (wrt. term positions) π' that has the properties stipulated above. By definition we know that $[\![t]\!]_{\mathcal{F}}|_{\pi'} = z$ and (since $\pi' < \pi$) also $[\![s]\!]_{\mathcal{F}}|_{\pi'} = z$. Since only positions strictly below π' are changed in the reduction step $s \overset{\pi}{\hookrightarrow}_{\mathcal{R},\mu} t$ and the subterms at π in both cleaned terms are equal, also $[\![s]\!]_{\mathcal{F}}$ and $[\![t]\!]_{\mathcal{F}}$ are equal. □

The next lemma states, that for left-linear TRSs termination is preserved when cleaning all terms to some signature that is an extension of the implicit signature of the TRS. This result is a generalization of Lemma 16 from [ST10].

Lemma 6.4.5. *Let $(\mathcal{R}, \mu)$ be a CS-TRS where $\mathcal{R}$ is left-linear. Let $\mathcal{F}$ be some signature such that $\mathcal{F} \supseteq \mathcal{F}(\mathcal{R})$. Then, if a term s over an arbitrary signature is terminating wrt. $(\mathcal{R}, \mu)$, also $[\![s]\!]_{\mathcal{F}}$ is terminating wrt. $(\mathcal{R}, \mu)$.*

Proof. This result is a direct consequence of the proof of Lemma 16 in [ST10]. We omit the generalization of this proof, because it is immediate from the mentioned proof. The base case works unchanged, and in the induction step the only non-trivial part, which also introduces the requirement for left-linearity, is when a reduction step at root position is considered. However, in this case context-sensitive rewriting does not differ from regular rewriting, so the proof works without changes. □

The next auxiliary result shows that for CS-DP problems where the rule system is left-linear, we may restrict attention to the signature of the DP problem without losing generality (This is again a generalized version of a lemma from [ST10]).

Lemma 6.4.6. *Let $(\mathcal{P}, \mathcal{R}, \mu)$ be a CS-DP problem where $\mathcal{R}$ is left-linear. Let $\mathcal{F}$ be some signature such that $\mathcal{F}_{>\epsilon}(\mathcal{P}) \cup \mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}$. Then, if there is an infinite minimal chain in $(\mathcal{P}, \mathcal{R}, \mu)$, there also is an infinite minimal chain in $(\mathcal{P}, \mathcal{R}, \mu)$ with terms over the signature $\mathcal{F}$.*

Proof. Suppose, that there is the following infinite sequence:

$$s_1 \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} t_1 \hookrightarrow_{\mathcal{R},\mu} s_2 \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} t_2 \hookrightarrow_{\mathcal{R},\mu} \dots$$

We define a new cleaning operation, that cleans only strictly below the root position of a term:

$$\langle\!\langle s \rangle\!\rangle_{\mathcal{F}} = \begin{cases} x & \text{if } s = x \in X \\ f([\![s_1]\!]_{\mathcal{F}}, \dots, [\![s_n]\!]_{\mathcal{F}}) & \text{if } s = f(s_1, \dots, s_n) \end{cases}$$

We claim, that

$$\langle\!\langle s_1 \rangle\!\rangle_{\mathcal{F}} \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} \langle\!\langle t_1 \rangle\!\rangle_{\mathcal{F}} \hookrightarrow^*_{\mathcal{R},\mu} \langle\!\langle s_2 \rangle\!\rangle_{\mathcal{F}} \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} \langle\!\langle t_2 \rangle\!\rangle_{\mathcal{F}} \hookrightarrow^*_{\mathcal{R},\mu} \dots$$

is an infinite minimal sequence as well. To see that this is true, first consider an arbitrary step $s_i \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} t_i$ in the assumed sequence. Since the step is applied at root position, we know that there is some rule $l \to r \in \mathcal{P}$ and some substitution σ such that $\sigma(l) = s_i$ and $\sigma(r) = t_i$. From $\sigma(l) = s_i$ we obtain $\langle\!\langle \sigma(l) \rangle\!\rangle_{\mathcal{F}} = \langle\!\langle s_i \rangle\!\rangle_{\mathcal{F}}$ and since we know that l does not contain any symbol not in $\mathcal{F}$ below the root, we get $[\![\sigma]\!]_{\mathcal{F}}(l) = \langle\!\langle s_i \rangle\!\rangle_{\mathcal{F}}$. Applying the same argument on $\sigma(r) = t_i$ yields $[\![\sigma]\!]_{\mathcal{F}}(r) = \langle\!\langle t_i \rangle\!\rangle_{\mathcal{F}}$. Thus we can conclude that $\langle\!\langle s_i \rangle\!\rangle_{\mathcal{F}} \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} \langle\!\langle t_i \rangle\!\rangle_{\mathcal{F}}$ is valid.
Now we consider an arbitrary sequence $t_i \hookrightarrow^*_{\mathcal{R},\mu} s_{i+1}$ that occurs in the assumed infinite minimal sequence. Take again an arbitrary step $t \hookrightarrow_{\mathcal{R},\mu} s$ from $t_i \hookrightarrow^*_{\mathcal{R},\mu} s_{i+1}$ (if there is one, otherwise the claim trivially holds). Since t_i was obtained by some $\mathcal{P}$-step applied at root position, we know that t_i is rooted by some symbol F from $\mathcal{F}^{(\#)}$. We also know, that t and s are rooted by the same symbol F, because starting from t_i both terms were reached only by using $\mathcal{R}$-steps and $\mathcal{R}$ does not contain any of the symbols in $\mathcal{F}^{(\#)}$. Let $l \to r \in \mathcal{R}$ be the rule used in the examined reduction step and let $\pi = i.\pi'$ be the position, at which the rule was applied on t. The terms s and t have the following structure:

$$t = F(u_1, \dots, u_{i-1}, u_i, u_{i+1}, \dots, u_n) \overset{\pi}{\hookrightarrow}_{\mathcal{R},\mu} F(u_1, \dots, u_{i-1}, v_i, u_{i+1}, \dots, u_n) = s$$

For $u_i \overset{\pi'}{\hookrightarrow}_{\mathcal{R},\mu} v_i$, Lemma 6.4.4 states, that either $[\![u_i]\!]_{\mathcal{F}} \hookrightarrow_{\mathcal{R},\mu} [\![v_i]\!]_{\mathcal{F}}$ or $[\![u_i]\!]_{\mathcal{F}} = [\![v_i]\!]_{\mathcal{F}}$ holds. Therefore, we obtain either

$$\begin{array}{rcl} \langle\!\langle t \rangle\!\rangle_{\mathcal{F}} & = & F([\![u_1]\!]_{\mathcal{F}}, \dots, [\![u_{i-1}]\!]_{\mathcal{F}}, [\![u_i]\!]_{\mathcal{F}}, [\![u_{i+1}]\!]_{\mathcal{F}}, \dots, [\![u_n]\!]_{\mathcal{F}}) \\ & & \hookrightarrow_{\mathcal{R},\mu} \\ & & F([\![u_1]\!]_{\mathcal{F}}, \dots, [\![u_{i-1}]\!]_{\mathcal{F}}, [\![v_i]\!]_{\mathcal{F}}, [\![u_{i+1}]\!]_{\mathcal{F}}, \dots, [\![u_n]\!]_{\mathcal{F}}) \quad = \quad \langle\!\langle s \rangle\!\rangle_{\mathcal{F}} \end{array}$$

or

$$\langle\!\langle t \rangle\!\rangle_{\mathcal{F}} = F([\![u_1]\!]_{\mathcal{F}}, \dots, [\![u_{i-1}]\!]_{\mathcal{F}}, [\![u_i]\!]_{\mathcal{F}}, [\![u_{i+1}]\!]_{\mathcal{F}}, \dots, [\![u_n]\!]_{\mathcal{F}}) = \langle\!\langle s \rangle\!\rangle_{\mathcal{F}}$$

Applying this result iteratively, we can construct a (possibly shorter) sequence $\langle\!\langle t_i \rangle\!\rangle_{\mathcal{F}} \hookrightarrow^*_{\mathcal{R},\mu} \langle\!\langle s_{i+1} \rangle\!\rangle_{\mathcal{F}}$ from $t_i \hookrightarrow^*_{\mathcal{R},\mu} s_{i+1}$. This concludes the proof that the new sequence is infinite. It remains to show, that it is also minimal. This is easy, because by assumption we know that every term t_i is terminating wrt. $(\mathcal{R}, \mu)$, and $\mathcal{R}$ is left-linear. By Lemma 6.4.5 it follows, that also every term $[\![t_i]\!]_{\mathcal{F}}$ is terminating wrt. $(\mathcal{R}, \mu)$, and since all steps in $(\mathcal{R}, \mu)$-rewrite sequences starting at t_i are applied strictly below the root, also $\langle\!\langle t_i \rangle\!\rangle_{\mathcal{F}}$ is terminating wrt. $(\mathcal{R}, \mu)$. □

Lemma 6.4.7. *Let $(\mathcal{P}, \mathcal{R}, \mu)$ be a CS-DP problem where $\mathcal{R}$ is left-linear. If the CS-DP problem $\mathcal{FC}_{\mathcal{F}}(\mathcal{P}, \mathcal{R}, \mu)$ is finite, then also $(\mathcal{P}, \mathcal{R}, \mu)$ is finite.*

Proof. Suppose, $(\mathcal{P}, \mathcal{R}, \mu)$ is not finite. Then there is an infinite minimal sequence in $(\mathcal{P}, \mathcal{R}, \mu)$. By Lemma 6.4.6 we know that without loss of generality we can assume, that all symbols occurring below root position are from $\mathcal{F}_{>\epsilon}(\mathcal{P}) \cup \mathcal{F}(\mathcal{R})$. Let $s_1 \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} t_1 \hookrightarrow^*_{\mathcal{R},\mu} s_2 \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} t_2 \hookrightarrow^*_{\mathcal{R},\mu} \dots$ be such a sequence. Then we claim that

$$\mathtt{block}(s_1) \overset{\epsilon}{\hookrightarrow}_{\mathtt{block}(\mathcal{P}),\mu_\Delta} \mathtt{block}(t_1) \hookrightarrow^*_{\mathcal{FC}_1(\mathcal{R}),\mu_\Delta} \mathtt{block}(s_2) \overset{\epsilon}{\hookrightarrow}_{\mathtt{block}(\mathcal{P}),\mu_\Delta} \dots$$

is an infinite minimal sequence wrt. $\mathcal{FC}_{\mathcal{F}}(\mathcal{P}, \mathcal{R}, \mu) = (\mathtt{block}(\mathcal{P}), \mathcal{FC}_{\mathcal{F}\cup\{\Delta\}}(\mathcal{R}, \mu_\Delta), \mu_\Delta)$.

Consider an arbitrary step $s_i \overset{\epsilon}{\hookrightarrow}_{\mathcal{P},\mu} t_i$ in the original sequence. By construction of $\mathtt{block}(\mathcal{P})$ and μ_Δ, the following step is also valid: $\mathtt{block}(s_i) \overset{\epsilon}{\hookrightarrow}_{\mathtt{block}(\mathcal{P}),\mu_\Delta} \mathtt{block}(t_i)$ (because we assumed, that function symbols from $\mathcal{F}^{(\#)}$ can only occur at root position).
Now, consider an an arbitrary $(\mathcal{R}, \mu)$-sequence $t_i \hookrightarrow^*_{\mathcal{R},\mu} s_{i+1}$. Let $s \hookrightarrow_{\mathcal{R},\mu} t$ be an arbitrary step within this sequence and let $l \to r \in \mathcal{R}$ be the rule applied at position π on the term s to obtain t. Since the reduction must take place below root position, we can write

$$s = F(u_1, \dots, u_{j-1}, u_j, u_{j+1}, \dots, u_k) \overset{\pi}{\hookrightarrow}_{l\to r,\mu} F(u_1, \dots, u_{j-1}, v_j, u_{j+1}, \dots, u_k) = t$$

Let π' be a position such that $\pi = j.\pi'$. Then we have $u_j \overset{\pi'}{\hookrightarrow}_{l\to r,\mu} v_j$. We have to distinguish two cases:

- $l \to r \in \mathcal{R}_p$: Then the rule $l \to r$ is also in $\mathcal{FC}_{\mathcal{F}\cup\{\Delta\}}(\mathcal{R}, \mu_\Delta)$ and since $\mu_\Delta(\Delta) = \{1\}$, this rule can be applied at position $j.1.\pi'$ on the term

  $$\mathtt{block}(s) = F(\Delta(u_1), \dots, \Delta(u_{j-1}), \Delta(u_j), \Delta(u_{j+1}), \dots, \Delta(u_k))$$

  in order to obtain $\mathtt{block}(t)$.

- $l \to r \notin \mathcal{R}_p$: Consider two cases again:
  - $\pi = j$ (i.e. $\pi' = \epsilon$): The rule $\Delta(l) \to \Delta(r)$ is in $\mathcal{FC}_{\mathcal{F}\cup\{\Delta\}}(\mathcal{R}, \mu_\Delta)$ (because $1 \in \mu_\Delta(\Delta)$) and this rule can be applied at position π on $\mathtt{block}(s)$ to obtain $\mathtt{block}(t)$.
  - $\pi > j$: Let π'' be a (possibly empty) position and n be an integer such that $\pi = j.\pi''.n$. Let $g = \mathtt{root}(s|_{j.\pi''})$. We know, that $g(y_1, \dots, y_{n-1}, l, y_{n+1}, \dots, y_m) \to g(y_1, \dots, y_{n-1}, r, y_{n+1}, \dots, y_m)$ is in $\mathcal{FC}_{\mathcal{F}\cup\{\Delta\}}(\mathcal{R}, \mu_\Delta)$ because, since $j.\pi''.n$ is an allowed position in s, we have $n \in \mu_\Delta(g)$. This rule can be applied at position $j.1.\pi''$ on the term $\mathtt{block}(s)$ to obtain $\mathtt{block}(t)$.

Thus, we can construct an infinite $\mathcal{FC}_{\mathcal{F}}(\mathcal{P}, \mathcal{R}, \mu)$ sequence from an infinite $(\mathcal{P}, \mathcal{R}, \mu)$ sequence. We still need to show, that the constructed sequence is also minimal, i.e. all $\mathtt{block}(t_i)$ are

terminating wrt. $(\mathcal{FC}_{\mathcal{F}\cup\{\Delta\}}(\mathcal{R},\mu_\Delta),\mu_\Delta)$. Suppose $\mathtt{block}(t_i)$ is not terminating for some i. Then there is an infinite rewrite sequence

$$\mathtt{block}(t_{i1}) \hookrightarrow_{\mathcal{FC}_{\mathcal{F}\cup\{\Delta\}}(\mathcal{R},\mu_\Delta),\mu_\Delta} \mathtt{block}(t_{i2}) \hookrightarrow_{\mathcal{FC}_{\mathcal{F}\cup\{\Delta\}}(\mathcal{R},\mu_\Delta),\mu_\Delta} \cdots$$

Since by construction of $\mathcal{FC}_{\mathcal{F}\cup\{\Delta\}}(\mathcal{R},\mu_\Delta)$, each step performed with a rule from this set can also be performed with a rule from $\mathcal{R}$ (at the same position or one position above), this implies $\mathtt{block}(t_{i1}) \hookrightarrow_{\mathcal{R},\mu} \mathtt{block}(t_{i2}) \hookrightarrow_{\mathcal{R},\mu} \ldots$, which in turn implies $t_{i1} \hookrightarrow_{\mathcal{R},\mu} t_{i2} \hookrightarrow_{\mathcal{R},\mu} \ldots$ because $\mathcal{R}$ does not contain the symbol Δ. This is a contradiction to the assumed minimality of the original sequence. Therefore, $\mathcal{FC}_\mathcal{F}(\mathcal{P},\mathcal{R},\mu)$ is not finite whenever $(\mathcal{P},\mathcal{R},\mu)$ is not finite. □

Remark. Note, that we do not need flat-context stability for the TRS $\mathcal{R}$ in $(\mathcal{P},\mathcal{R},\mu)$ for the above translation, as it is automatically enforced by the introduction of the symbol Δ.

Definition 6.4.8 (Root-Labeling Transformation for CS-DP Problems, [SM08])**.** Let $(\mathcal{P},\mathcal{R},\mu)$ be a DP problem and let $\mathcal{F}^{(\#)}$ and $\mathcal{F}$ be as in Definition 6.1.18 and let Δ and μ_Δ be as in Definition 6.4.1. Let $\mathcal{F}' = \mathcal{F} \cup \mathcal{F}^{(\#)} \cup \{\Delta\}$. Let $\mathcal{P}'$ be the TRS with the rules from $\mathtt{block}(\mathcal{P})$, over the signature $\mathcal{F}'$ and let $\mathcal{R}'$ be the TRS with rules from $\mathcal{FC}_{\mathcal{F}\cup\{\Delta\}}(\mathcal{R},\mu_\Delta)$ over signature $\mathcal{F}'$. The root-labeling transformation $\mathcal{FC}_\mathcal{F}(\mathcal{P},\mathcal{R},\mu)_{\mathtt{rl}}$ is defined as the triple $(\mathcal{P}'_{\mathtt{rl}},\mathcal{R}'_{\mathtt{rl}},\mu_{\mathtt{rl}})$, where the root-labeling operation is modified in the following way: for each function symbol F in $\mathcal{F}^{(\#)}$, the set of labels is $S_F = \{F\}$, the label mapping is $\pi_F = F$ and the interpretation of F in $\mathcal{A}_{\mathcal{F}'}$ is $F_{\mathcal{A}_{\mathcal{F}'}}(x_1,\ldots,x_n) = g$ for all $x_1,\ldots,x_n \in \mathcal{F}'$ and an arbitrary but fixed $g \in \mathcal{F}^{(\#)}$. Furthermore, for all labeled function symbols f_l occurring in $\mathcal{P}'_{\mathtt{rl}}$ and $\mathcal{R}'_{\mathtt{rl}}$, let $\mu_{\mathtt{rl}}(f_l) = \mu(f)$, where f is the unlabeled counterpart of f_l.

Theorem 6.4.9. *Let $(\mathcal{P},\mathcal{R},\mu)$ be a CS-DP problem. The following CS-DP processor is sound:*

$$(\mathcal{P},\mathcal{R},\mu) \mapsto \begin{cases} \{\mathcal{FC}_\mathcal{F}(\mathcal{P},\mathcal{R},\mu)_{\mathtt{rl}}\} & \text{if } \mathcal{R} \text{ is left-linear} \\ \{(\mathcal{P},\mathcal{R},\mu)\} & \text{otw.} \end{cases}$$

Proof. According to Lemma 6.4.7, whenever $\mathcal{FC}_\mathcal{F}(\mathcal{P},\mathcal{R},\mu)$ is finite, also $(\mathcal{P},\mathcal{R},\mu)$ is finite because $\mathcal{R}$ is left-linear. As root-labeling is a special case of Theorem 6.1.20, we know that finiteness of $\mathcal{FC}_\mathcal{F}(\mathcal{P},\mathcal{R},\mu)$ is equivalent to finiteness of $\mathcal{FC}_\mathcal{F}(\mathcal{P},\mathcal{R},\mu)_{\mathtt{rl}}$. This concludes the soundness proof, as the DP problem $\mathcal{FC}_\mathcal{F}(\mathcal{P},\mathcal{R},\mu)_{\mathtt{rl}}$ is finite if and only if $\mathcal{FC}_\mathcal{F}(\mathcal{P},\mathcal{R},\mu)$ is finite, which in turn is finite only if $(\mathcal{P},\mathcal{R},\mu)$ is finite. □

6.5 Implementation for VMTL

The direct method version of root-labeling is implemented in the class `RootLabeling` in the package `dpvis.logic.dm.methods`. The DP processor is implemented in the class `RootLabelingProcessor` in the package `dpvis.logic.dp`.

The Direct Method

Parameters

The direct method version of root-labeling supports some parameters that can be used to limit the size of the transformed systems.

Parameter	Values	Description
`max_output_rules`	$\mathbb{N}_0$ Default: 0	Sets a threshold for the maximum number of rules that the transformed system may contain. The number of rules that the transformed system will contain is calculated before the transformation is started. If the number exceeds this value, the processor fails immediately and returns the unchanged input system. If this value is zero, the number of rules in the transformed system is unrestricted.
`max_output_growth`	$\mathbb{N}_0$ Default: 0	Sets a threshold for the maximum number of rules that the transformed system may contain. This value specifies the maximal allowed percentage of increase in the number of rules, compared to the original system. If the number exceeds this value, the processor fails immediately and returns the unchanged input system. If this value is zero, the number of rules in the transformed system is unrestricted.
`max_arity`	$\mathbb{N}_0$ Default: 0	Sets a threshold for the maximum arity that any function symbol in the signature of the original system may have. If there are symbols with a greater arity, the processor fails immediately and returns the unchanged input system. If this value is zero, the maximum arity is not restricted.

The (CS-)DP Processor

Parameters

The (CS-) DP processor provides the same parameters as the direct method. In addition, the maximum number of dependency pairs in the translated DP problem can be specified.

Parameter	Values	Description
`max_output_rules`	$\mathbb{N}_0$ Default: 0	Sets a threshold for the maximum number of rules that the transformed DP problem may contain. The number of rules that the transformed DP problem will contain is calculated before the transformation is started. If the number exceeds this value, the processor fails immediately and returns the unchanged input problem. If this value is zero, the number of rules in the transformed problem is unrestricted.
`max_output_dps`	$\mathbb{N}_0$ Default: 0	Sets a threshold for the maximum number of dependency pairs that the transformed DP problem may contain. The number of dependency pairs that the transformed DP problem will contain is calculated before the transformation is started. If the number exceeds this value, the processor fails immediately and returns the unchanged input problem. If this value is zero, the number of dependency pairs in the transformed problem is unrestricted.
`max_output_growth`	$\mathbb{N}_0$ Default: 0	Sets a threshold for the maximum number of rules and dependency pairs that the transformed DP problem may contain. This value specifies the maximum allowed percentage of increase in the number of rules and dependency pairs, compared to the respective numbers in the input problem. If one of the numbers exceeds this value, the processor fails immediately and returns the unchanged input problem. If this value is zero, the number of rules and dependency pairs in the transformed problem is unrestricted.
`max_arity`	$\mathbb{N}_0$ Default: 0	Sets a threshold for the maximum arity that any function symbol in the signature of the original DP problem may have. If there are symbols with a greater arity, the processor fails immediately and returns the unchanged input problem. If this value is zero, the maximum arity is not restricted.

The implementation of root-labeling for VMTL translates (CS-) DP problems (or TRSs, respectively) as described in this chapter. There is one minor difference however: For consistency with the definition of semantic labeling, we required that root-labeling assigns labels for constant symbols as well. Constant symbols always get themselves as their label. For example the symbol c would always be labelled as $c_{(c)}$. Since same constant symbols are always labeled in the same way, the label can be left implicit. This is done in the implemented versions of root-labeling, where constant symbols do not get any label.

Internally, function symbols are just strings in VMTL. Whenever VMTL generates an HTML output, labels are displayed subscripted (e.g. $f_{(a,b,a)}(a, b, x)$). This is done by the output processing of VMTL. Internally, the term from the example would be stored as $\texttt{f\_\{a.b.a\}}(\texttt{a}, \texttt{b}, \texttt{x})$. Usually, this raw form is hidden from the user when working with HTML output. However, for example when debugging VMTL, terms are output in this form.

Example outputs can be found in the appendix in Section B.2.

CHAPTER 7

Knuth-Bendix Order

One part of this thesis is the implementation of the Knuth-Bendix-Order (KBO) for use with VMTL. It should be implemented both as a direct method and as a CS-DP processor. The implementation mostly closely follows the approach described by Zankl, Hirokawa and Middeldorp in [ZHM09], where the search for a KBO that can be used to prove termination of a given TRS is modelled as a SAT problem.

7.1 Knuth-Bendix Order

The Knuth-Bendix order was developed by Donald Knuth and Peter Bendix in 1970 and is presented in their paper "Simple Word Problems in Universal Algebra" [KB70] where it was used as part of a completion algorithm. Today different, (mostly) equivalent versions of the Knuth-Bendix order are used. Here, we give the definition used in [ZHM09]. Knuth-Bendix orders use two parameters: A *weight function* and an ordering relation on function symbols. As an ordering relation, we use a quasi-ordering $\succsim$ (i.e. a transitive and reflexive relation). The strict part of $\succsim$ is denoted as $\succ$ and the equivalence part is denoted as $\sim$.

Definition 7.1.1 (Weight Function, Admissibility)**.** A *weight function* is a pair (w, w_0), where $w_0 \in \mathbb{N}$ and w is a mapping $w : \mathcal{F} \mapsto \mathbb{N}_0$ such that $w(c) \geq w_0$ for all $c \in \mathcal{F}^{(0)}$. A weight function is called *admissible with respect to a quasi ordering* $\succsim$ if for all $f \in \mathcal{F}$, $w(f) = 0$ implies $f \succsim g$ for all $g \in \mathcal{F}$.

The weight function can be extended to a function from terms to natural numbers. To this end, every variable gets assigned the weight w_0 and the weight of a term is the sum of the weights of all variables and the weights of all function symbols occurring in the term:

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

Definition 7.1.2 (Knuth-Bendix Order, [KB70, ZHM09])**.** Let (w, w_0) be a weight function, and let $\succsim$ be a quasi-precedence. The *Knuth-Bendix order* $>_{\text{kbo}}$ is defined on terms $s, t \in \mathcal{T}(\mathcal{F}, X)$ by: $s >_{\text{kbo}} t$ iff $|s|_x \geq |t|_x$ for all $x \in X$ and either:

(KBO1) $w(s) > w(t)$ or

(KBO2) $w(s) = w(t)$ and one of:

(KBO2a) $t \in X, s \in \mathcal{T}(\mathcal{F}^{(1)}, \{t\})$ and $s \neq t$

(KBO2b) $s = f(s_1, \ldots, s_n), t = g(t_1, \ldots, t_m)$ and $f \succ g$

(KBO2c) $s = f(s_1, \ldots, s_n), t = g(t_1, \ldots, t_m)$ and $f \sim g$ and there is an index i with $1 \leq i \leq \min\{m, n\}$ and $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$ and $s_i >_{\text{kbo}} t_i$

Here, $\succ$ denotes the strict part of the quasi order $\succsim$ and $\sim$ denotes the equivalence part of the quasi order $\succsim$. The strict part of a quasi order $\succsim$ is defined as $x \succ y$ iff $x \succsim y \wedge \neg(y \succsim x)$. The equivalence part of a quasi-order $\succsim$ is defined as $x \sim y$ iff. $x \succsim y \wedge y \succsim x$. A different version of KBO uses a strict order $>$ instead of $\succsim$. In this case, in (KBO2c) instead of $f \sim g$, $f = g$ is required.

7.2 Implementation as a Direct Method

The following result from [ZHM09] describes how KBO can be used to prove termination of a TRS. Using this lemma, proving termination of a TRS with KBO amounts to finding a quasi-precedence and an admissible weight function.

Theorem 7.2.1 ([ZHM09])**.** *Let $\mathcal{R} = (\mathcal{F}, R)$ be a TRS. $\mathcal{R}$ is terminating whenever there exists a quasi-precedence $\succsim$ and an admissible weight function (w, w_0) such that $R \subseteq >_{kbo}$.*

Example 7.2.2. Consider the following TRS $\mathcal{R}$, for which termination should be proven with the help of KBO:

$$\begin{array}{lcl@{\qquad\qquad}lcl} \mathsf{plus}(\mathsf{0},x) & \to & x & \mathsf{plus}(\mathsf{s}(x),y) & \to & \mathsf{s}(\mathsf{plus}(x,y)) \\ \mathsf{plus}(x,\mathsf{0}) & \to & x & \mathsf{plus}(x,\mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{plus}(x,y)) \end{array}$$

Now, consider the KBO induced by the following parameters:

- For the quasi-ordering we require $\mathsf{plus} \succ \mathsf{s}$.
- For the weight-function, we define $w_0 = 1$, $w(\mathsf{0}) = 1$ and $w(\mathsf{s}) = w(\mathsf{plus}) = 2$.

Note, that the weight-function is admissible wrt. the quasi-ordering $\succsim$. This KBO orients all rules in $\mathcal{R}$:

- The rule $\mathsf{plus}(\mathsf{0},x) \to x$ is orientable because for this rule the condition KBO1 is satisfied, since the weight of $\mathsf{plus}(\mathsf{0},x)$ is strictly greater than the weight of x ($w(\mathsf{plus}(\mathsf{0},x)) = 4, w(x) = 1$).

- For the rule $\mathsf{plus}(x, 0) \to x$, also the condition KBO1 is satisfied for the same reason.
- For the rule $\mathsf{plus}(\mathsf{s}(x),y) \to \mathsf{s}(\mathsf{plus}(x,y))$ the condition KBO2 is satisfied, because both sides have the same weight. Additionally, the condition KBO2b is satisfied, because we defined $\mathsf{plus} \succ \mathsf{s}$.
- For the last rule, $\mathsf{plus}(x, \mathsf{s}(y)) \to \mathsf{s}(\mathsf{plus}(x,y))$, also KBO2 and KBO2b are satisfied again.

By Theorem 7.2.1, we know that $\mathcal{R}$ is terminating, because the constructed KBO orients all rules, and the weight-function is admissible with respect to the quasi-ordering.

The search for a quasi-precedence $\succsim$ and an admissible weight function (w, w_0) that induce a KBO, which can be used to prove termination of a TRS $\mathcal{R}$, is modelled as a SAT problem in [ZHM09]. The implementation of the KBO as a direct method for VMTL follows the definitions in [ZHM09]. This section summarizes these definitions.

Specifying the Problem Formula

We use the notation of $\mathcal{T}$-formulas (cf. Chapter 5) for the SMT-formulas presented here. In the following, bit-vector variables are written in bold-face with the length of the bit-vector as subscript. Bit-vectors will be used for specifying arithmetic calculations and relations as part of the SAT/SMT problem (e.g. calculating term weights). In the actual implementation, real or integer values can also be used. This can be done in the straightforward way, by just using the respective values or variables instead of the bit-vector values or variables.

The weight function (w, w_0) is encoded as a bit-vector variable $(\mathbf{w}_0)_k$ representing w_0 and a set $\mathbf{w} = \{\mathbf{f}_k \mid f \in \mathcal{F}\}$ containing a bit-vector variable for each function symbol in $\mathcal{F}$, that represents the weight of the corresponding function symbol. The quasi precedence $\succsim$ is represented by means of the two sets $X = \{X_{fg} \mid f, g \in \mathcal{F}\}$ and $Y = \{Y_{fg} \mid f, g \in \mathcal{F}\}$ of propositional variables, where for a model $\mathcal{M}$, $\mathcal{M} \models X_{fg}$ means, that in the solution encoded by $\mathcal{M}$, $f \succ g$ holds, and $\mathcal{M} \models Y_{fg}$ means $f \sim g$ holds.

The formula ADM-SAT$_k(w, w_0)$ enforces, that the weight function encoded by a model is indeed a weight function, and also that it is an admissible weight function with respect to the encoded quasi-precedence $\succsim$. That is, if a model $\mathcal{M}$ satisfies ADM-SAT$_k(w, w_0)$, the weight function encoded by $\mathcal{M}$ is admissible wrt. $\succsim$.

Definition 7.2.3 (ADM-SAT$_k(w, w_0)$, [ZHM09])**.** The formula ADM-SAT$_k(w, w_0)$ is defined as:

$$(\mathbf{w}_0)_k > \mathbf{0}_k \wedge \bigwedge_{c \in \mathcal{F}^{(0)}} \mathbf{c}_k \geq (\mathbf{w}_0)_k \wedge \bigwedge_{f \in \mathcal{F}^{(1)}} \left(\mathbf{f}_k = \mathbf{0}_k \to \bigwedge_{g \in \mathcal{F}} (X_{fg} \vee Y_{fg}) \right)$$

The weight of terms is calculated by the function W_k^t mapping terms t to (sums of) bit-vector variables representing the weight of t.

Definition 7.2.4 (W_k^t, [ZHM09])**.** W_k^t is inductively defined as:

$$\mathrm{W}_k^t = \begin{cases} (\mathbf{w}_0)_k & \text{if } t \in X \\ \mathbf{f}_k + \sum_{i=1}^{n} \mathrm{W}_k^{t_i} & \text{if } t = f(t_1, \ldots, t_n). \end{cases}$$

The sets X and Y are intended to encode the quasi ordering $\succsim$. However, we need to enforce, that the relation encoded by X and Y is indeed a quasi-ordering on $\mathcal{F}$. In [ZHM09], this is done by assigning to each function symbol in $\mathcal{F}$ a fresh bit-vector variable and requiring, that in each model the following formula $\mathrm{QUASI}_j(\succsim)$ is satisfied.

Definition 7.2.5 ($\mathrm{QUASI}_j(\succsim)$)**.**

$$\mathrm{QUASI}_j(\succsim) = \bigwedge_{f,g \in \mathcal{F}} \big((X_{fg} \to \mathbf{f}'_j > \mathbf{g}'_j) \wedge (Y_{fg} \to \mathbf{f}'_j = \mathbf{g}'_j) \big)$$

The formulas $\mathrm{SAT}_k(s >_{\text{kbo}} t)$ and $\mathrm{SAT}_k(s >'_{\text{kbo}} t)$ are used to encode the KBO conditions (KBO1), (KBO2) and (KBO2a)-(KBO2c), More precisely, in every model $\mathcal{M}$ such that $\mathcal{M} \models \mathrm{SAT}_k(s >_{\text{kbo}} t)$, we have, that $|s|_x \geq |t|_x$ for all $x \in V$ and either (KBO1) holds for s and t or (KBO2) holds for s and t and additionally (encoded via $\mathrm{SAT}_k(s >'_{\text{kbo}} t)$), (KBO2a), (KBO2b) or (KBO2c) hold for s and t.

Definition 7.2.6 ($\mathrm{SAT}_k(s >_{\text{kbo}} t)$, [ZHM09])**.** Let s and t be terms. The formula $\mathrm{SAT}_k(s >_{\text{kbo}} t)$ is defined as follows. If $s \in X$ or $s = t$ or $|s|_x < |t|_x$ for some $x \in X$, then $\mathrm{SAT}_k(s >_{\text{kbo}} t) = \bot$. Otherwise

$$\mathrm{SAT}_k(s >_{\text{kbo}} t) = \mathrm{W}_k^s > \mathrm{W}_k^t \vee (\mathrm{W}_k^s = \mathrm{W}_k^t \wedge \mathrm{SAT}_k(s >'_{\text{kbo}} t))$$

with

$$\mathrm{SAT}_k(s >'_{\text{kbo}} t) = \begin{cases} \top & \text{if } t \in X, s \in \mathcal{T}(\mathcal{F}^{(1)}, \{t\}), \text{and } s \neq t \\ X_{fg} \vee (Y_{fg} \wedge \mathrm{SAT}_k(s_i >_{\text{kbo}} t_i) & \text{if } s = f(s_1, \ldots, s_n), t = g(t_1, \ldots, t_m) \end{cases}$$

where in the second clause, i denotes the least value i such that $1 \leq i \leq \min\{n, m\}$ and $s_i \neq t_i$.

Putting the defined formulas together, we obtain the formula $\text{KBO-SAT}_{k,j}(\mathcal{R})$, which is satisfiable for some values k and j if $\mathcal{R}$ can be oriented with a KBO.

Definition 7.2.7 ($\text{KBO-SAT}_{k,j}(\mathcal{R})$, [ZHM09])**.**

$$\text{ADM-SAT}_k(w, w_0) \wedge \mathrm{QUASI}_j(\succsim) \wedge \bigwedge_{l \to r \in \mathcal{R}} \mathrm{SAT}_k(l >_{\text{kbo}} r)$$

Solving the Problem Formula via SAT/SMT Solving

For SAT and SMT solving, we have to choose a word length for the bit-vector variables that are used. In [ZHM09] it is shown that an upper bound $B_{\mathcal{R}}$ for symbol weights can be computed, such that if termination of a TRS $\mathcal{R}$ can be shown with a KBO, then there is a KBO where all symbol weights are less than or equal to $B_{\mathcal{R}}$. Thus, the word length required for the bit-vector variables encoding the weight function in the problem formula needs to be at most $\lceil \log_2(B_{\mathcal{R}} + 1) \rceil$. For the bit-vector variables used to enforce the properties of a quasi-ordering, a word-length of at least $\lceil \log_2 n \rceil$ is sufficient, where n is the number of function symbols in the signature of $\mathcal{R}$.

Theorem 7.2.8 (Termination criterion, [ZHM09])**.** *Termination of a TRS $\mathcal{R}$ can be shown by KBO whenever the problem formula KBO-SAT$_{k,j}(\mathcal{R})$ is satisfiable for some $k \in \mathbb{N}$ and $j = \lceil \log_2 n \rceil$, where n is the number of function symbols in the signature of $\mathcal{R}$.*

If $k \geq \lceil \log_2(B_{\mathcal{R}} + 1) \rceil$ then also the other direction works.

For SAT solvers, the used bit-vectors have to be transformed into propositional logic. This is done implicitly by the VSSF if a SAT solver is used as the back-end solver.

7.3 Implementation as a (CS-)DP Processor

As a DP processor (resp.. CS-DP processor), KBO will be used in a reduction pair processor (CS-reduction pair processor) with argument filtering and usable rules. Due to the definitions and results in [AEF$^+$08] the adaption of a reduction pair processor to a context sensitive reduction pair processor can be done in a very straightforward way for the biggest part. The only part requiring some more consideration is the computation of usable rules, as they cannot simply be generalized for the context-sensitive case. Therefore, the processor implemented for VMTL will make a case distinction on the input problem, and act either as a (non-CS) reduction pair processor or a CS-reduction pair processor accordingly.

The Reduction Pair Processor

Reduction pairs and argument filterings [AG00, GAO02] can be used in a DP processor (called "Reduction Pair Processor") to simplify the set of dependency pairs in a DP problem.

Definition 7.3.1 (Reduction Pair [AG00, GAO02])**.** A *reduction pair* is a pair $(\succ, \succsim)$ where

- $\succ$ is a well-founded order that is closed under substitutions and
- $\succsim$ is a quasi-order that is closed under substitutions and contexts and
- $\succsim \circ \succ \circ \succsim \subseteq \succ$ (i.e. $\succsim$ and $\succ$ are compatible).

Given a DP problem $(\mathcal{P}, \mathcal{R})$, if it is possible to orient all dependency pairs strictly or weakly (i.e. $\mathcal{P} \subseteq \succsim \cup \succ$), and orient all rules weakly (i.e. $\mathcal{R} \subseteq \succsim$), then $(\mathcal{P}, \mathcal{R})$ is finite iff $(\mathcal{P} \setminus \succ, \mathcal{R})$

is finite. However, it is not necessary to prove weak orientability for *all* rules in $\mathcal{R}$. It suffices to show that the set of *usable rules* of $(\mathcal{P}, \mathcal{R})$ is weakly orientable. To reduce the number of usable rules further, argument filterings can be used.

Definition 7.3.2 (Argument Filtering [AG00, KNT99])**.** An argument filtering for a signature $\mathcal{F}$ is a mapping π from $\mathcal{F}$ to the natural numbers or (possibly empty) lists of natural numbers such that for each $n \geq 0$ and $f \in \mathcal{F}^{(n)}$ either

- $\pi(f) = k$ for some $k \in \mathbb{N}$ with $1 \leq k \leq n$ or
- $\pi(f) = [k_1, k_2, \ldots, k_m]$ for $k_1, k_2, \ldots, k_m \in \mathbb{N}$ and $1 \leq k_1 < k_2 < \cdots < k_m \leq n$

Argument filterings can be applied to terms, to obtain new terms. Let $t \in \mathcal{T}(\mathcal{F}, X)$ be a term and π an argument filtering for $\mathcal{F}$. Then $\pi(t)$ is defined as follows:

$$\pi(t) = \begin{cases} t & \text{if } t \in X \\ \pi(t_i) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } \pi(f) = i \\ f(\pi(t_{i_1}), \ldots, \pi(t_{i_m})) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } \pi(f) = [i_1, \ldots, i_m] \end{cases}$$

This mapping maps terms from $\mathcal{T}(\mathcal{F}, X)$ to terms from $\mathcal{T}(\mathcal{F}_\pi, X)$ where in the signature $\mathcal{F}_\pi$ the same function symbols are contained, but might have lower arity. For each $f \in \mathcal{F}$, if $\pi(f) \in \mathbb{N}$ then in $\mathcal{F}_\pi$, the function symbol f has arity 0 and if $\pi(f) = [k_1, k_2, \ldots, k_m]$, then in $\mathcal{F}_\pi$ the function symbol f has arity m.

Example 7.3.3. As an example, consider the following TRS $\mathcal{R}$ which exends the TRS from Example 7.2.2 by two additional rules:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathsf{plus}(\mathsf{0},x) & \to & x & \mathsf{plus}(\mathsf{s}(x),y) & \to & \mathsf{s}(\mathsf{plus}(x,y)) \\ \mathsf{plus}(x,\mathsf{0}) & \to & x & \mathsf{plus}(x,\mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{plus}(x,y)) \\ \mathsf{times}(x,\mathsf{0}) & \to & \mathsf{0} & \mathsf{times}(x,\mathsf{s}(y)) & \to & \mathsf{plus}(\mathsf{times}(x,y),x) \end{array}$$

Consider the argument filtering π where $\pi(\mathsf{plus}) = [1]$, $\pi(\mathsf{times}) = 2$ and $\pi(\mathsf{s}) = 1$. Applying the argument filtering to the lhs and rhs of each rule yields the following new TRS:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathsf{plus}(\mathsf{0}) & \to & x & \mathsf{plus}(x) & \to & \mathsf{plus}(x) \\ \mathsf{plus}(x) & \to & x & & & \\ \mathsf{0} & \to & \mathsf{0} & y & \to & \mathsf{plus}(y) \end{array}$$

For an ordering relation $>$ on terms from $\mathcal{T}(\mathcal{F}, X)$, the relation $>_\pi$ is defined as $s >_\pi t \Leftrightarrow \pi(s) > \pi(t)$, for all $s, t \in \mathcal{T}(\mathcal{F}, X)$.

Argument filterings reduce the number of usable rules that have to be oriented weakly in the reduction pair processor. In [GTSKF06] the usable rules with respect to an argument filtering are defined. Here, we split calculation of usable rules into two parts, first calculating the so-called usable symbols, which then directly give rise to the usable rules. This two step calculation will be necessary in the SAT encoding of the reduction pair processor based on KBO.

Definition 7.3.4 (Regarded Positions, [GTSKF06])**.** Let π be an argument filtering. For any n-ary function symbol f, the set $\mathtt{rp}_\pi(f)$ of regarded positions is $\{i\}$ if $\pi(f) = i$ and it is $\{i_1, \ldots, i_m\}$ if $\pi(f) = [i_1, \ldots, i_m]$.

Definition 7.3.5 (Usable Rules wrt. Argument Filtering, [GTSKF06])**.** Let $\mathcal{R}$ be a TRS over the signature $\mathcal{F}$ and let $f \in \mathcal{F}$. The set $\mathtt{Rls}_\mathcal{R}(f)$ is defined as $\{l \to r \in \mathcal{R} \mid \mathtt{root}(l) = f\}$. Let $F \subseteq \mathcal{F}$ be a set of function symbols. Then $\mathtt{Rls}_\mathcal{R}(F) = \bigcup_{f \in F} \mathtt{Rls}_\mathcal{R}(f)$. Let $t \in \mathcal{T}(\mathcal{F}, X)$ be a term and π be an argument filtering. The set $\mathcal{US}_\mathcal{R}(t, \pi)$ of usable symbols with respect to π is defined as follows:

$$\begin{array}{rcll} \mathcal{US}_\mathcal{R}(x, \pi) & = & \emptyset & \text{for } x \in X \\ \mathcal{US}_\mathcal{R}(f(t_1, \ldots, t_n), \pi) & = & f & \cup \bigcup\limits_{l \to r \in \mathtt{Rls}_R(f)} \mathcal{US}_{\mathcal{R} \setminus \mathtt{Rls}_R(f)}(r, \pi) \cup \\ & & & \bigcup\limits_{i \in \mathtt{rp}_\pi(f)} \mathcal{US}_{\mathcal{R} \setminus \mathtt{Rls}_R(f)}(t_i, \pi) \end{array}$$

For a DP problem $(\mathcal{P}, \mathcal{R})$, the set of usable rules wrt. an argument filtering π is defined as

$$\mathcal{U}_\mathcal{R}(\mathcal{P}, \pi) = \bigcup_{s \to t \in \mathcal{P}} \mathtt{Rls}_R(\mathcal{US}_\mathcal{R}(t, \pi))$$

Example 7.3.6. Consider the TRS $\mathcal{R}$ from Example 7.3.3. The corresponding DP problem $(DP(\mathcal{R}), \mathcal{R})$ is:

Dependency Pairs:

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{plus}^\#(x, \mathsf{s}(y)) & \to & \mathsf{plus}^\#(x,y) & \mathsf{plus}^\#(\mathsf{s}(x),y) & \to & \mathsf{plus}^\#(x,y) \\ \mathsf{times}^\#(x, \mathsf{s}(y)) & \to & \mathsf{plus}^\#(\mathsf{times}(x,y),x) & \mathsf{times}^\#(x, \mathsf{s}(y)) & \to & \mathsf{times}^\#(x,y) \end{array}$$

Rules:

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{plus}(\mathsf{0},x) & \to & x & \mathsf{plus}(\mathsf{s}(x),y) & \to & \mathsf{s}(\mathsf{plus}(x,y)) \\ \mathsf{plus}(x, \mathsf{0}) & \to & x & \mathsf{plus}(x, \mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{plus}(x,y)) \\ \mathsf{times}(x, \mathsf{0}) & \to & \mathsf{0} & \mathsf{times}(x, \mathsf{s}(y)) & \to & \mathsf{plus}(\mathsf{times}(x,y),x) \end{array}$$

Let π be an argument filtering where $\pi(\mathsf{plus}) = [1]$, $\pi(\mathsf{plus}^\#) = []$, $\pi(\mathsf{times}) = 2$, $\pi(\mathsf{times}^\#) = [1]$ and $\pi(\mathsf{s}) = 1$. Applying this argument filtering to the lhs and rhs of all dependency pairs and all rules of the DP problem $(DP(\mathcal{R}), \mathcal{R})^\pi$ yields the following new DP problem:

Dependency Pairs:

$$\begin{array}{rcl} \mathsf{plus}^{\#} & \to & \mathsf{plus}^{\#} \\ \mathsf{times}^{\#}(x) & \to & \mathsf{times}^{\#}(x) \end{array} \qquad\qquad \mathsf{times}^{\#}(x) \to \mathsf{plus}^{\#}$$

Rules:

$$\begin{array}{rcl} \mathsf{plus}(\mathsf{0}) & \to & x \\ \mathsf{plus}(x) & \to & x \\ \mathsf{0} & \to & \mathsf{0} \end{array} \qquad\qquad \begin{array}{rcl} \mathsf{plus}(x) & \to & \mathsf{plus}(x) \\ y & \to & \mathsf{plus}(y) \end{array}$$

Since none of the right-hand-sides of any of the dependency pairs contains any symbol that is defined in the set of rules, none of the rules are usable. Therefore, we get: $\mathcal{U}_{\mathcal{R}}(DP(\mathcal{R}), \pi) = \emptyset$.

For the definition of the reduction pair processor, we need the notion of $\mathcal{C}_\epsilon$-compatibility for relations.

Definition 7.3.7 ($\mathcal{C}_\epsilon$-Compatibility, [GTSKF06])**.** A relation $\succsim$ is called $\mathcal{C}_\epsilon$-compatible if for a new symbol c, $c(x, y) \succsim x$ and $c(x, y) \succsim y$ holds.

However, $\mathcal{C}_\epsilon$-compatibility is just a formal requirement and will not concern us in the following, because KBO is a *simplification order*, and for these orders $\mathcal{C}_\epsilon$-compatibility is always trivially satisfied.

Definition 7.3.8 (Reduction Pair Processor, [GTSKF06])**.** Let $(\succ, \succsim)$ be a reduction pair such that $\succsim$ is $\mathcal{C}_\epsilon$-compatible and let π be an argument filtering. The reduction pair processor `Proc` is:

$$\texttt{Proc}((\mathcal{P}, \mathcal{R})) = \begin{cases} \{(\mathcal{P} \setminus \succ_\pi, \mathcal{R})\} & \text{if } \mathcal{P} \subseteq (\succ_\pi \cup \succsim_\pi), \mathcal{P} \cap \succ_\pi \neq \emptyset \text{ and } \mathcal{U}_{\mathcal{R}}(\mathcal{P}, \pi) \subseteq \succsim_\pi \\ \{(\mathcal{P}, \mathcal{R})\} & \text{otherwise} \end{cases}$$

The DP processor `Proc` is sound and complete [GTSKF06].

As a concrete example of a reduction pair processor, we can use KBOs to construct reduction pairs. Let $>_{\text{kbo}}$ be as in Definition 7.1.2 and let $\geq_{\text{kbo}}$ be a relation on terms, such that for two terms s and t, $s \geq_{\text{kbo}} t$ holds iff $s = t$ or $s >_{\text{kbo}} t$, where $=$ means syntactic equivalence. Furthermore, we require that the KBO $>_{\text{kbo}}$ uses an admissible weight-function wrt. its quasi-precedence. Then $(>_{\text{kbo}}, \geq_{\text{kbo}})$ is a valid reduction pair:

- We know, that any KBO that is induced by some weight function w and quasi-ordering $\succsim$ is a simplification order if w is admissible wrt. $\succsim$ [BN98]. Therefore, we know that $>_{\text{kbo}}$ is well-founded and closed under substitutions.
- We know further, that $\geq_{\text{kbo}}$ is closed under substitutions and under contexts (because already $>_{\text{kbo}}$ is closed under substitutions and under contexts, again because it is a simplification order).
- Obviously the two orders are compatible.

Example 7.3.9. Consider the DP problem $(DP(\mathcal{R}), \mathcal{R})$ and the argument filtering π from Example 7.3.6. As we found out in Example 7.3.6, none of the rules of this DP problem are usable. Therefore, we need to find some reduction pair, such that all dependency pairs in $(DP(\mathcal{R}), \mathcal{R})$ can be oriented weakly, and at least one of the dependency pairs in $(DP(\mathcal{R}), \mathcal{R})$ can be oriented strictly.

The following dependency pairs have to be oriented weakly resp. strictly:

$$\begin{array}{rcl} \mathsf{plus}^{\#} & \to & \mathsf{plus}^{\#} \\ \mathsf{times}^{\#}(x) & \to & \mathsf{times}^{\#}(x) \end{array} \qquad\qquad \mathsf{times}^{\#}(x) \to \mathsf{plus}^{\#}$$

Now, consider the KBO $>_{\text{kbo}}$ with a weight function w where $w_0 = 1$ and $w(\mathsf{times}^{\#}) = 2$ and $w(\mathsf{plus}^{\#}) = 2$. Using the reduction pair $(>_{\text{kbo}}, \geq_{\text{kbo}})$ as defined before, we can orient the dependency pairs $\mathsf{plus}^{\#} \to \mathsf{plus}^{\#}$ and $\mathsf{times}^{\#}(\mathsf{x}) \to \mathsf{times}^{\#}(\mathsf{x})$ weakly using $\geq_{\text{kbo}}$ and we can orient the dependency pair $\mathsf{times}^{\#}(\mathsf{x}) \to \mathsf{plus}^{\#}$ strictly using $>_{\text{kbo}}$. Thus, given the DP problem $(DP(\mathcal{R}), \mathcal{R})$ and argument filtering π, the result of a reduction pair processor, using the reduction pair $(>_{\text{kbo}}, \geq_{\text{kbo}})$, would be the DP problem $(\mathcal{P}, \mathcal{R})$, where $\mathcal{P} = DP(\mathcal{R}) \setminus \{\mathsf{times}^{\#}(\mathsf{x}, \mathsf{s}(\mathsf{y})) \to \mathsf{times}^{\#}(\mathsf{x}, \mathsf{y})\}$.

The Context-Sensitive Reduction Pair Processor

The adaption of the reduction pair processor with usable rules to the CS-DP framework is presented in [AEF$^+$08]. The notion of reduction pairs can be generalized, by requiring only μ-monotonicity for the relation $\succsim$. As in the context-free case, the use of usable rules reduces the number of rules that have to be oriented weakly by $\succsim$. However, the straightforward generalization of usable rules to the context-sensitive case only works for CS-DP problems that are strongly conservative (see below). For CS-DP problems that are not strongly conservative, a different, weaker version of the calculation of usable rules is required.

Definition 7.3.10 (μ-Monotonicity)**.** A binary relation $\circ$ on terms is called *μ-monotonic*, if whenever for $s, t \in \mathcal{T}(\mathcal{F}, X)$, $s \circ t$ holds also

$$f(t_1, \ldots, t_{i-1}, s, t_{i+1}, \ldots, t_n) \circ f(t_1, \ldots, t_{i-1}, t, t_{i+1}, \ldots, t_n)$$

holds for all $f \in \mathcal{F}^{(n)}$, $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n \in \mathcal{T}(\mathcal{F}, X)$ and $i \in \mu(f)$.

Definition 7.3.11 (μ-reduction pair, [AEF$^+$08])**.** A *μ-reduction pair* is a pair $(\succ, \succsim)$ where

- $\succ$ is a well-founded order that is closed under substitutions and
- $\succsim$ is a μ-monotonic quasi-order that is closed under substitutions and
- $\succsim$ and $\succ$ are compatible.

For the calculation of usable rules, [AEF$^+$08] presents two versions. The stronger version requires some additional assumptions about the set of rules, that are not required for the weaker version.

Definition 7.3.12 (CS-Usable Rules, [AEF+08]). Let $\mathtt{Rls}_{\mathcal{R}}(f) = \{l \to r \in \mathcal{R} \mid \mathtt{root}(l) = f\}$. For any symbols f and h and CS-TRS $(\mathcal{R}, \mu)$, let:

- $f \blacktriangleright_{\mathcal{R},\mu} h$ if
  - $f = h$ or
  - there is a symbol g with $g \blacktriangleright_{\mathcal{R},\mu} h$ and a rule $l \to r \in \mathcal{R}$ with $g \in \mathtt{Func}^{\mu}(r)$.
- $f \triangleright_{\mathcal{R},\mu} h$ if
  - $f = h$ or
  - there is a symbol g with $g \triangleright_{\mathcal{R},\mu} h$ and a rule $l \to r \in \mathcal{R}$ with $g \in \mathtt{Func}^{\bar{\mu}}(l) \cup \mathtt{Func}^{\mu}(r)$.

The two versions of usable symbols are

$$\mathcal{US}^{\blacktriangleright}(\mathcal{P}, \mathcal{R}, \mu) = \bigcup_{\substack{s \to t \in \mathcal{P} \\ f \in \mathtt{Func}^{\mu}(t) \\ f \blacktriangleright_{\mathcal{R},\mu} g}} g$$

$$\mathcal{US}^{\triangleright}(\mathcal{P}, \mathcal{R}, \mu) = \bigcup_{\substack{s \to t \in \mathcal{P} \\ f \in \mathtt{Func}^{\bar{\mu}}(s) \cup \mathtt{Func}(t) \\ f \triangleright_{\mathcal{R},\mu} g}} g \cup \bigcup_{\substack{l \to r \in \mathcal{R} \\ f \in \mathtt{Func}^{\bar{\mu}}(r) \\ f \triangleright_{\mathcal{R},\mu} g}} g$$

which yield the following sets of usable rules:

$$\mathcal{U}^{\blacktriangleright}(\mathcal{P}, \mathcal{R}, \mu) = \mathtt{Rls}_{\mathcal{R}}(\mathcal{US}^{\blacktriangleright}(\mathcal{P}, \mathcal{R}, \mu))$$
$$\mathcal{U}^{\triangleright}(\mathcal{P}, \mathcal{R}, \mu) = \mathtt{Rls}_{\mathcal{R}}(\mathcal{US}^{\triangleright}(\mathcal{P}, \mathcal{R}, \mu))$$

For the stronger definition of usable rules ($\mathcal{U}^{\blacktriangleright}(\mathcal{P}, \mathcal{R}, \mu)$), we need the notion of strongly conservative CS-TRSs, where some restrictions on the occurrences of variables hold.

Definition 7.3.13 (Strongly Conservative [AEF+08]). A CS-TRS $(\mathcal{R}, \mu)$ is called *strongly conservative* if for all rules $l \to r \in \mathcal{R}$ the following conditions hold:

- $\mathtt{Vars}^{\mu}(r) \subseteq \mathtt{Vars}^{\mu}(l)$
- $\mathtt{Vars}^{\mu}(l) \cap \mathtt{Vars}^{\bar{\mu}}(l) = \emptyset$
- $\mathtt{Vars}^{\mu}(r) \cap \mathtt{Vars}^{\bar{\mu}}(r) = \emptyset$

Definition 7.3.14 (CS-Reduction Pair Processor, [AEF+08]). Let $(\succ, \succsim)$ be a μ-reduction pair where $\succsim$ is $\mathcal{C}_{\epsilon}$-compatible. For a CS-DP problem $(\mathcal{P}, \mathcal{R}, \mu)$, the result of $\mathtt{Proc}((\mathcal{P}, \mathcal{R}, \mu))$ is:

- $\{(\mathcal{P} \setminus \succ, \mathcal{R}, \mu)\}$, if $\mathcal{P} \subseteq (\succ \cup \succsim)$ and at least one of the following holds:
  - $\mathcal{U}^{\blacktriangleright}(\mathcal{P}, \mathcal{R}, \mu) \subseteq \succsim$, $\mathcal{P} \cup \mathcal{U}^{\blacktriangleright}(\mathcal{P}, \mathcal{R}, \mu)$ is strongly conservative

– $\mathcal{U}^{\triangleright}(\mathcal{P}, \mathcal{R}, \mu) \subseteq \succsim$

- $\{(\mathcal{P}, \mathcal{R}, \mu)\}$ otherwise.

The CS-DP processor `Proc` is sound [AEF+08].

A SAT/SMT Encoding

To build a reduction pair processor, we are looking for a strict order $>$ and an admissible weight function (w, w_0), inducing a KBO $>_{\text{kbo}}$. Let $\geq_{\text{kbo}}$ be defined as before. The search for the two parameters inducing such a KBO is again encoded as a SAT/SMT problem in [ZHM09]. We mostly closely follow the approach presented there.

Much of the complexity of the formulas encoding the DP processor comes from the use of argument filterings. The argument filtering is encoded as part of the SAT problem. Argument filterings can substantially change the structure of terms. Therefore, many things that could previously (when defining the direct method) be determined statically, now need to be encoded as part of the SAT problem as well. An example is syntactic equivalence of terms. Previously, syntactic equivalence of terms could be determined on a purely static level (e.g. by using standard term comparison in Java). However, when argument filterings are used, syntactic equivalence becomes dependent of the encoded argument filtering, so now syntactic equivalence of terms has to be expressed in terms of a propositional formula as part of the SAT problem. This makes the SAT problem considerably bigger. Therefore, for performance reasons, it should be possible to disable the use of argument filterings if necessary.

The DP processor implemented here is (like all other DP processors that are implemented for VMTL) actually a CS-DP processor. The processor makes a case distinction and checks, whether the input problem is a DP problem or a CS-DP problem and continues accordingly. There are some substantial differences between the two processors:

- The calculation of usable rules is different (see above).
- The definition of the CS-DP processor with usable rules from [AEF+08] we use here does not support argument filterings.

Since the CS-DP processor does not use argument filterings, the use of argument filterings is automatically disabled when context-sensitive dependency pairs are treated. Therefore, in this case the calculation of usable rules does not need to be done as part of the SAT problem. Instead, the restriction to usable rules is done statically, before the SAT encoding is constructed.

The set $\{\pi_f \mid f \in \mathcal{F}\} \cup \{\pi_f^i \mid f \in \mathcal{F}, 1 \leq i \leq \mathtt{ar}(f)\}$ of propositional variables is used to encode the argument filtering for the context-free case. Let $\mathcal{M}$ be a model. The argument filtering π encoded by $\mathcal{M}$ is determined as follows. If $\mathcal{M}$ does not satisfy π_f for some symbol f, and satisfies π_f^i, then $\pi(f) = i$. On the other hand, if $\mathcal{M}$ satisfies π_f for some symbol f, and $\mathcal{M}$ satisfies the variables $\{\pi_f^{i_1}, \ldots, \pi_f^{i_m}\} \subseteq \{\pi_f^i \mid f \in \mathcal{F}, 1 \leq i \leq \mathtt{ar}(f)\}$ and does not satisfy the variables $\{\pi_f^i \mid f \in \mathcal{F}, 1 \leq i \leq \mathtt{ar}(f)\} \setminus \{\pi_f^{i_1}, \ldots, \pi_f^{i_m}\}$, then $\pi(f) = [i_1, \ldots, i_m]$. The following formula from [ZHM09] enforces the encoding to be valid (i.e. if for some $f \in \mathcal{F}$, π_f

evaluates to false under a model satisfying $\mathrm{AF}^\pi(f)$, then π_f^i evaluates to true for exactly one i with $1 \leq i \leq \mathtt{ar}(f)$, otherwise π_f^i may be true for arbitrarily many i with $1 \leq i \leq \mathtt{ar}(f)$).

Definition 7.3.15 ($\mathrm{AF}^\pi(f)$, [ZHM09])**.**

$$\mathrm{AF}^\pi(f) = \pi_f \vee \bigvee_{i=1}^{\mathtt{ar}(f)} \left(\pi_f^i \wedge \bigwedge_{j \neq i} \neg \pi_f^j \right)$$

Applied on a signature $\mathcal{F}$, the formula $\mathrm{AF}^\pi(\mathcal{F})$ is defined as $\bigwedge_{f \in \mathcal{F}} \mathrm{AF}^\pi(f)$.

In the encoding of KBO as a direct method, syntactic equivalence of terms was statically determined at the time the SAT problem was formulated. This is no longer the case for the DP processor implementation, because equivalence of terms is now always modulo an argument filtering, which is determined through the encoding in the SAT problem. Therefore, the formula $s =^\pi t$ is introduced in [ZHM09], which is satisfiable if and only if $\pi(s) = \pi(t)$ for terms s and t (The formula is stated here, as defined in [ZHM09], but in the implementation it is optimized as suggested in the same paper).

Definition 7.3.16 ($s =^\pi t$, [ZHM09])**.** Let s and t be terms in $\mathcal{T}(\mathcal{F}, X)$. The propositional formula $s =^\pi t$ is defined by induction on s and t. If $s \in X$ then

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

Let $s = f(s_1, \ldots, s_n)$. If $t \in X$, then

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

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

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

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

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

Another property that was statically determined at the absence of argument filterings is non-duplication of a pair of terms. The formula $\mathrm{ND}_j^\pi(s,t)$ from [ZHM09] (slightly reformulated here, as a VSSF-formula) asserts $|\pi(s)|_x \geq |\pi(t)|_x$ for terms s and t under the encoded argument filtering π.

Definition 7.3.17 ($\mathrm{ND}^{\pi}_{j}(s,t)$, [ZHM09])**.**

$$\mathrm{ND}^{\pi}_{j}(s,t) = \bigwedge_{x \in \mathtt{Vars}(t)} |s|^{j}_{x} \geq |t|^{j}_{x}$$

with

$$|s|^{j}_{x} = \begin{cases} \mathbf{1}_j & \text{if } s = x \\ \mathbf{0}_j & \text{if } s \in X, s \neq x \\ \sum_{i=1}^{n}(|s_i|^{j}_{x} \times \pi^{i}_{f}) & \text{if } s = f(s_1, \ldots, s_n) \end{cases}$$

Calculation of term weights works similarly as before, but again, the encoded argument filtering has to be considered. The expression $\mathrm{w}^{\pi}_{k}(t)$ yields a bit-vector variable that represents the weight of the term t under the encoded argument filtering π. Again, the formula from [ZHM09] is reformulated slightly as a VSSF-formula.

Definition 7.3.18 ($\mathrm{w}^{\pi}_{k}(t)$, [ZHM09])**.**

$$\mathrm{w}^{\pi}_{k}(t) = \begin{cases} (\mathbf{w}_0)_k & \text{if } s = x \\ \mathbf{f}_k \times \pi_f + \sum_{i=1}^{n}(\mathrm{w}^{\pi}_{k}(t_i) \times \pi^{i}_{f}) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

The next formula $\mathrm{SAT}_k(s >^{\pi}_{\mathrm{kbo}} t)$ is satisfiable iff under the encoded weight function, quasi-precedence and argument filtering, the terms s and t can be strictly oriented with the KBO. The weak version $\mathrm{SAT}_k(s \geq^{\pi}_{\mathrm{kbo}} t)$ is satisfiable iff $\mathrm{SAT}_k(s >^{\pi}_{\mathrm{kbo}} t)$ is satisfiable or $s =^{\pi} t$ is satisfiable.

Definition 7.3.19 ($\mathrm{SAT}_k(s >^{\pi}_{\mathrm{kbo}} t)$, [ZHM09])**.**

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

For the weak version we have: $\mathrm{SAT}_k(s \geq^{\pi}_{\mathrm{kbo}} t) = \mathrm{SAT}_k(s >^{\pi}_{\mathrm{kbo}} t) \vee s =^{\pi} t$ The formula $\mathrm{SAT}_k(s >^{\pi}_{\mathrm{kbo}}{}' t)$ is defined by induction on the structure of s and t. If $s \in X$ then $\mathrm{SAT}_k(s >^{\pi}_{\mathrm{kbo}}{}' t) = \bot$. If $s = f(s_1, \ldots, s_n)$ and $t \in X$, then

$$\mathrm{SAT}_k(s >^{\pi}_{\mathrm{kbo}}{}' t) = \pi_f \wedge \bigvee_{i=1}^{n}(\pi^{i}_{f} \wedge \mathrm{SAT}_k(s_i \geq^{\pi}_{\mathrm{kbo}}{}' t)) \vee \neg\pi_f \wedge \bigvee_{i=1}^{n}(\pi^{i}_{f} \wedge \mathrm{SAT}_k(s_i >^{\pi}_{\mathrm{kbo}}{}' t))$$

where $\mathrm{SAT}_k(s \geq^{\pi}_{\mathrm{kbo}}{}' t) = \mathrm{SAT}_k(s >^{\pi}_{\mathrm{kbo}}{}' t) \vee s =^{\pi} t$.
If $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ with $f \neq g$ then

$$\begin{aligned} \mathrm{SAT}_k(s >^{\pi}_{\mathrm{kbo}}{}' t) \quad = \quad & (\pi_f \wedge \pi_g \wedge X_{fg}) \vee \\ & \vee \neg\pi_g \wedge \textstyle\bigvee_{j=1}^{m}(\pi^{j}_{g} \wedge \mathrm{SAT}_k(s >^{\pi}_{\mathrm{kbo}} t_j)) \vee \\ & \vee \neg\pi_f \wedge \textstyle\bigvee_{i=1}^{n}(\pi^{i}_{f} \wedge \mathrm{SAT}_k(s_i >^{\pi}_{\mathrm{kbo}} t)) \end{aligned}$$

If $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$ then

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

where $\langle s_1, \ldots, s_n\rangle >^{\pi,f}_{\text{kbo}} \langle t_1, \ldots, t_n\rangle = \bot$ if $n = 0$ and otherwise

$$\begin{aligned}\langle s_1, \ldots, s_n\rangle >^{\pi,f}_{\text{kbo}} \langle t_1, \ldots, t_n\rangle \quad = \quad & (\pi^1_f \wedge \text{SAT}_k(s_1 >^{\pi}_{\text{kbo}} t_1)) \vee \\ & \vee(\pi^1_f \rightarrow s_1 =^{\pi} t_1 \wedge \langle s_2, \ldots, s_n\rangle >^{\pi,f}_{\text{kbo}} \langle t_2, \ldots, t_n\rangle).\end{aligned}$$

The formula $\text{ADM}^{\pi}_k(\mathcal{F})$ from [ZHM09] is used to assert admissibility of the encoded weight function. It is satisfiable if and only if the encoded weight function is admissible with respect to the encoded quasi-ordering and the encoded argument filtering. To take into account that the argument filtering may change the arity of function symbols, the formulas $\text{constant}^{\pi}(f)$ and $\text{unary}^{\pi}(f)$ are introduced, being satisfiable if and only if f is constant (resp. unary) in the signature $\mathcal{F}_{\pi}$ for the encoded argument filtering π.

Definition 7.3.20 ($\text{ADM}^{\pi}_k(\mathcal{F})$, [ZHM09])**.**

$$\begin{aligned}\text{ADM}^{\pi}_k(\mathcal{R}) \quad = \quad & (\mathbf{w}_0)_k > \mathbf{0}_k \wedge \\ & \bigwedge_{f\in\mathcal{F}} (\text{constant}^{\pi}(f) \rightarrow \mathbf{f} \geq (\mathbf{w}_0)_k) \wedge \\ & \bigwedge_{f\in\mathcal{F}} \left(\mathbf{f} = \mathbf{0} \wedge \text{unary}^{\pi}(f) \rightarrow \bigwedge_{g\in\mathcal{F}, f\neq g} (\pi_g \rightarrow X_{fg}) \right)\end{aligned}$$

where

$$\text{constant}^{\pi}(f) = \pi_f \wedge \bigwedge_{i=1}^{\mathtt{ar}(f)} \neg\pi^i_f$$

and

$$\text{unary}^{\pi}(f) = \pi_f \wedge \bigvee_{i=1}^{\mathtt{ar}(f)} \left(\pi^i_f \wedge \bigwedge_{i\neq j} \neg\pi^j_f \right)$$

As in the direct method encoding (where a quasi-ordering was used), it is necessary to make sure that the encoded relation $>$ is really a strict order. This is achieved by satisfying the formula $\text{PO}(>)$ [ZHM09].

Definition 7.3.21 ($\text{PO}^{\pi}_j(>)$, [ZHM09])**.**

$$\text{PO}^{\pi}_j(>) = \bigwedge_{f,g\in\mathcal{F}} X_{fg} \rightarrow \mathbf{f}'_j > \mathbf{g}'_j$$

One technique making reduction pair processors very strong are usable rules. For a DP problem $(\mathcal{P}, \mathcal{R})$ (respectively a CS-DP problem $(\mathcal{P}, \mathcal{R}, \mu)$) the usable rules are a subset of the rules in $\mathcal{R}$, that have to be shown weakly orientable with respect to the order $\succsim$ of the reduction pair used in the reduction pair processor. The calculation of usable rules depends on whether the problem that should be processed is a DP problem or a CS-DP problem. In [ZHM09], the encoding of the calculation of usable rules for DP problems is presented. To this end, a set $\left\{\mathrm{U}_f \mid f \in \mathcal{F}^{\#}\right\}$ of propositional variables is introduced. Let $\mathcal{M}$ be a model. The semantics of a variable U_f is, that if under the argument filtering encoded in $\mathcal{M}$, a rule $f(l_1, \ldots, l_n) \to r \in \mathcal{R}$ for a DP problem $(\mathcal{P}, \mathcal{R})$ is usable, then U_f is satisfied in $\mathcal{M}$.

The formula given in [ZHM09] combines calculation of usable rules, and assertion of orientability. Here, we split the calculation of usable rules and orientability into distinct formulas. The formula $\mathrm{US}^{\pi}((\mathcal{P}, \mathcal{R}))$ (which is based on the formula from [ZHM09]) asserts, that (at least) those variables U_f are satisfied, that give rise to the usable rules of $(\mathcal{P}, \mathcal{R})$.

Definition 7.3.22 ($\mathrm{US}^{\pi}((\mathcal{P}, \mathcal{R}))$)**.** The formula $\mathrm{US}^{\pi}((\mathcal{P}, \mathcal{R}))$ asserts, that for each usable rule $f(l_1, \ldots, l_n) \to r \in \mathcal{R}$ wrt. $\mathcal{P}$ and the encoded argument filtering π, the propositional variable U_f is satisfied.

$$\begin{aligned}\mathrm{US}^{\pi}((\mathcal{P}, \mathcal{R})) \quad = \quad & \bigwedge_{l \to r \in \mathcal{P}} \mathrm{U}_{\mathtt{root}(l)} \wedge \\ & \bigwedge_{l \to r \in \mathcal{P} \cup \mathcal{R}} \left(\mathrm{U}_{\mathtt{root}(l)} \to \bigwedge_{\substack{p \in \mathtt{FPos}(r) \\ \mathtt{root}(r|_p)\text{defined}}} \left(\left(\bigwedge_{\substack{q,i: \\ i.q \leq p}} \pi^i_{\mathtt{root}(r|_q)} \right) \to \mathrm{U}_{\mathtt{root}(r|_q)} \right) \right)\end{aligned}$$

The following formula takes care of the orientability of all dependency pairs and all usable rules. It enforces, that all dependency pairs and rules are weakly orientable and that at least one dependency pair is strictly orientable (in order to guarantee some progress – otherwise the SAT/SMT solver might find a model that does not help removing any dependency pair and might miss another model that does).

Definition 7.3.23 ($\mathrm{ORIENT}^{\pi}_k((\mathcal{P}, \mathcal{R}))$)**.**

$$\begin{aligned}\mathrm{ORIENT}^{\pi}_k((\mathcal{P}, \mathcal{R})) \quad = \quad & \bigwedge_{l \to r \in \mathcal{P}} \mathrm{SAT}_k(l \geq^{\pi}_{\mathrm{kbo}} r) \wedge \bigvee_{l \to r \in \mathcal{P}} \mathrm{SAT}_k(l >^{\pi}_{\mathrm{kbo}} r) \wedge \\ & \bigwedge_{l \to r \in \mathcal{R}} \left(\mathrm{U}_{\mathtt{root}(l)} \to \mathrm{SAT}_k(l \geq^{\pi}_{\mathrm{kbo}} r)\right)\end{aligned}$$

The context-sensitive DP processor as defined in Definition 7.3.14, which we are implementing, does not support argument filterings. Therefore, for the context-sensitive case the usable rules can be determined statically before the proof attempt is started. Furthermore, in this case, many of the formulas we defined above, can be replaced by simpler formulas, that do not need to consider the argument filtering variables. For example, the formula $\mathrm{ND}^{\pi}_j(s, t)$ just evaluates to $\top$ if the rule $s \to t$ is non-duplicating (ignoring the argument filtering π) and it evaluates to $\bot$ otherwise.

Putting everything together, we get two formulas encoding the DP processor and CS-DP processor, respectively. The DP processor is encoded by the formula $\text{RPP}_{k,j}((\mathcal{P},\mathcal{R}))$ defined as

$$\text{US}^\pi((\mathcal{P},\mathcal{R})) \wedge \text{ORIENT}^\pi_k((\mathcal{P},\mathcal{R})) \wedge \text{AF}^\pi(\mathcal{F}) \wedge \text{ADM}^\pi_k(\mathcal{F}) \wedge \text{PO}^\pi_j(>)$$

and the CS-DP processor is encoded by the formula $\text{CSRPP}_{k,j}((\mathcal{P},\mathcal{R},\mu))$ defined as

$$\text{ORIENT}^\pi_k((\mathcal{P},\mathcal{R})) \wedge \text{ADM}^\pi_k(\mathcal{F}) \wedge \text{PO}^\pi_j(>).$$

A model $\mathcal{M}$ satisfying the first formula for a DP problem $(\mathcal{P},\mathcal{R})$ encodes a strict order $>$ and a weight function (w, w_0), admissible wrt. $>$, inducing a strict KBO $>_{\text{kbo}}$ and a weak KBO $\geq_{\text{kbo}}$, as well as an argument filtering π with the following properties:

- For all rules and dependency pairs $l \to r \in \mathcal{U}_\mathcal{R}(\mathcal{P},\pi) \cup \mathcal{P}$ we have $\pi(l) \geq_{\text{kbo}} \pi(r)$.
- For at least one dependency pair $l \to r \in \mathcal{P}$ we have $\pi(l) >_{\text{kbo}} \pi(r)$.

Since $\geq_{\text{kbo}}$ is closed under contexts and substitutions and is $\mathcal{C}_\epsilon$-compatible and $>_{\text{kbo}}$ is closed under substitutions and well-founded, $(>_{\text{kbo}}, \geq_{\text{kbo}})$ is a reduction pair and can be used in the reduction pair processor.

The same is true for models satisfying the second formula for CS-DP problem $(\mathcal{P},\mathcal{R},\mu)$. Such a model yields a reduction pair that can be used to remove dependency pairs from $\mathcal{P}$.

7.4 Implementation for VMTL

In this section, we discuss the implementation of the KBO as a direct method and as a DP processor for VMTL. Both versions use the VSSF for specifying and solving the constructed SAT problems. Numeric values can be represented by means of bit-vectors, integers or real values respectively variables over these domains. For SAT-solvers, the VSSF automatically translates bit-vector formulas into pure propositional formulas, so pure SAT solvers can work on these problems as well.

The Direct Method

The direct method version of the KBO is implemented for VMTL in the class `KBO` which can be found in the package `dpvis.logic.dm.methods`. It can be used for both context-free and context-sensitive TRSs. Context-sensitive TRSs are treated by this method just like context-free TRSs.

Parameters

The KBO method provides the following parameters that can be chosen in the VMTL strategy file when using the KBO method:

Parameter	Values	Description
`print_statistics`	`true` `false` (default)	Chooses whether to include statistics about the proof attempt in the output or not.
`debug`	`true` `false` (default)	Chooses whether to include debug-information in the output or not. The debug-information may be interesting for other things than debugging as well, as it contains for example calculated term weights.
`word_length`	$\mathbb{N}_0$ Default: 5	Chooses the word-length that is used for the calculation of term weights (i.e. the number of bits used in bit-vectors and bit-vector variables that represent symbol- and term weights.

`backend`	`minisat`
	`clasp`
	`yices` (default)
		Selects the solver that should be used for solving the encoded SMT problem. MiniSat and CLASP are SAT solvers and Yices is a SMT solver. For the SAT solvers, the encoded SMT problem is automatically translated into a SAT-equivalent SAT problem.

`arithmetic_theory`	`bitvectors`
	`ints` (default)
	`reals`
		Selects the theory that should be used for expressing numeric values and arithmetic operations on these values. The system automatically checks, whether the chosen back-end solver supports the selected theory. If an incompatible back-end solver is chosen, the system automatically switches the arithmetic theory to "`bitvectors`".

Output

VMTL only includes output of direct methods, if they were successful. If the proof attempt was successful, the parameters for a KBO have been found that proves termination of the input system. The output contains the following information:

- The weight-function and quasi-precedence of the KBO that proves termination of the input system.
- If the `print_statistics`-parameter is set to `true`, some statistical information about the proof attempt. This includes:
  - The time needed for creating the problem specification.

  - The time needed for solving the specified problem with the chosen back-end solver.
  - The number of variables (bit-vectors and propositional) that were used in the specification.
  - The number of assertions (formulas) that were used in the specification.
  - The word-size used for the bit-vectors.
- If the `debug`-parameter is set to `true` the calculated term weights are printed as well.

The output is created as an HTML document by the VMTL `HTMLOutputWriter` class.

The (CS-)DP Processor

The DP processor version of the KBO is implemented for VMTL in the class `KBOProcessor` which can be found in the package `dpvis.logic.dp`. It can be used for both context-free and context-sensitive TRSs.

By default, this processor uses usable rules. The calculation of usable rules is different for context-sensitive dependency pairs. The processor automatically chooses the correct method for calculating usable rules. Usable rules are calculated at the beginning of the proof search and as part of the SMT problem (for the non-context-sensitive case). The use of usable rules can also be deactivated by setting the respective parameter of this processor.

The use of argument filterings can also be disabled. For the context-sensitive case, the use of argument filterings is automatically disabled. Disabling argument filterings can considerably speed up the proof search, because a simpler SMT-problem can be created. However, it will make the processor less powerful.

Parameters

The KBO processor provides the following parameters that can be chosen in the VMTL strategy file when using the KBO processor:

Parameter	Values	Description
`print_statistics`	`true` `false` (default)	Chooses whether to include statistics about the proof attempt in the output or not.
`debug`	`true` `false` (default)	Chooses whether to include debug-information in the output or not. The debug-information may be interesting for other things than debugging as well, as it contains for example calculated term weights.
`use_usable_rules`	`true` (default) `false`	Used to decide whether usable rules should be calculated or not.
`use_argument_filtering`	`true` (default) `false`	Used to decide whether usable argument filterings should be used or not.
`word_length`	$\mathbb{N}_0$ Default: 5	Chooses the word-length that is used for the calculation of term weights (i.e. the number of bits used in bit-vectors and bit-vector variables that represent symbol- and term weights.

`backend`	`minisat`
	`clasp`
	`yices` (default)
		Selects the solver that should be used for solving the encoded SMT problem. MiniSat and CLASP are SAT solvers and Yices is an SMT solver. For the SAT solvers, the encoded SMT problem is automatically translated into an SAT-equivalent SAT problem.

`arithmetic_theory`	`bitvectors`
	`ints` (default)
	`reals`
		Selects the theory that should be used for expressing numeric values and arithmetic operations on these values. The system automatically checks, whether the chosen back-end solver supports the selected theory. If an incompatible back-end solver is chosen, the system automatically switches the arithmetic theory to "`bitvectors`".

Output

VMTL only includes output of DP processors, if they were successful. If the proof attempt was successful, the parameters of a KBO are presented, which weakly orients all usable rules and dependency pairs and strictly orients at least one of the dependency pairs. The strictly oriented dependency pairs are removed. If all dependency pairs are removed, finiteness of the (CS-)DP problem has been proved. Otherwise, the resulting system is handed back to the VSSF framework which chooses the next processor that tries to show finiteness of the new problem.

The output of the KBO processor contains the following elements:

- The argument filtering π that was used (if argument filterings were used).
- The weight-function and strict order of a KBO that weakly orients all usable rules and all dependency pairs and strictly orients at least one dependency pair (all with respect to the argument filtering π).
- The argument-filtered version of the dependency pair problem, where usable rules are specially marked.
- The strictly oriented dependency pairs which are removed.
- If the `print_statistics`-parameter is set to `true`, some statistical information about the proof attempt. This includes:
  - The time needed for creating the problem specification.
  - The time needed for solving the specified problem with the chosen back-end solver.
  - The number of variables (bit-vectors and propositional) that were used in the specification.
  - The number of assertions (formulas) that were used in the specification.
  - The word-size used for the bit-vectors.
- If the `debug`-parameter is set to `true` the following additional information is printed:
  - The calculated term weights of all terms that occur as the left-hand side or right-hand side of a rule or dependency pair in the DP problem.
  - The number of bits that are used for bit-vectors representing the number of variable occurrences in argument-filtered terms. (Required to enforce non-duplication under argument filterings)
  - For each function symbol, whether the symbol is unary under the encoded argument filtering or not.

The output is created as an HTML document by the VMTL `HTMLOutputWriter` class.

CHAPTER 8

Tests and Benchmarks

In this chapter, we discuss how the new processors and methods were tested and present the results of these tests. Section 8.1 summarizes the results for the technique of KBO and Section 8.2 summarizes the results for the root-labeling technique. In Section 8.3, we present a new default strategy for VMTL that can be used for general purpose applications.

8.1 Knuth-Bendix-Order

The KBO method and processor were tested by using special strategies that use the KBO techniques and running VMTL on a set of 1584 problems from the termination database (version 4). The results were then compared with the results of the reference implementation from [ZHM09]. Cases that are not covered by the reference implementation or differ from the reference implementation (i.e. cases, where the reference implementation found "MAYBE" or "TIMEOUT" and the VMTL implementation found "YES") were checked manually.

The VMTL implementation of KBO supports various solvers and theories for the computation of numeric values. The tests were performed with two SMT solvers (Yices [DdM06] and z3 [dMB08]) and two SAT solvers (Minisat [SE05] and clasp [GKNS07]). For the SMT solvers, the numeric theories of integers and reals as well as differently sized bit-vectors were used. For the SAT solvers differently sized bit-vectors were used.

The Direct Method

For the tests of the direct method, we used a very simple strategy that just applies the KBO method as preprocessing and does not perform a DP-analysis.

Example 8.1.1. The following strategy file was used for the test of the KBO method, with Minisat as the SAT-solving back-end and encoding numeric values as bit-vectors of length 4.

```
<Strategy>
<Preprocessing>
   <KBO print_statistics="true" debug="true"
        word_length="4"
        arithmetic_theory="bitvectors"
        backend="minisat"/>
</Preprocessing>
</Strategy>
```

Listing 8.1: Strategy for the tests of the KBO method

For the other test cases, the parameters of the KBO method were modified accordingly in the strategy.

All in all, 18 different configurations were tested. The next table summarizes the results of the tests. For each test-case we present the number of TRSs in the set of test problems that could be proven to be terminating (YES) or non-terminating (NO). Furthermore, we show the number of problems where termination could not be decided (MAYBE) or the computation exceeded the time-limit of 60 seconds (T/O). The table also contains the average execution time of the KBO method.

Solver	Theory	YES	NO	MAYBE	T/O	Avg. KBO Time
Yices	Integers	116	60	1408	0	36.30 ms
Yices	Reals	116	60	1408	0	36.30 ms
Yices	BV/2	72	60	1452	0	40.31 ms
Yices	BV/3	116	60	1408	0	38.79 ms
Yices	BV/4	116	60	1408	0	41.21 ms
Yices	BV/5	116	60	1408	0	42.70 ms
Yices	BV/6	116	60	1408	0	43.56 ms
Minisat	BV/4	116	60	1408	0	108.37 ms
Minisat	BV/5	116	60	1408	0	175.76 ms
Minisat	BV/6	116	60	1406	2	380.84 ms
clasp	BV/4	116	60	1408	0	134.22 ms
clasp	BV/5	116	60	1408	0	166.39 ms
clasp	BV/6	116	60	1406	2	319.44 ms
z3	Integers	116	60	1408	0	56.61 ms
z3	Reals	116	60	1408	0	56.45 ms
z3	BV/4	116	60	1408	0	79.12 ms
z3	BV/5	116	60	1407	1	139.60 ms
z3	BV/6	116	60	1404	4	268.80 ms

The original implementation from [ZHM09] was tested by the authors on a set of 1381

problems. The results of their tests can be found online[1]. We compared our results to their results, for checking if our method works as expected.

In the tests, the VMTL version of the KBO method could prove termination of the same problems that could be shown to be terminating by the reference implementation. Additionally, in our tests we considered some problems from the TPDB, that were not considered in the tests of the reference implementation. These cases were checked manually if termination was found. All solutions were found to be correct.

When using bit-vectors for the representation of numeric values, a size of 3 bits suffices to prove termination of all TRSs in our test-set, that could be proven to be terminating with stronger theories like integers or reals. The tests showed, that the Yices SMT-solver in combination with the Integers- or Reals-theory works most efficiently, with an average execution time of the KBO method of 36.30 ms, proving termination of 116 TRSs and producing no timeouts.

The DP Processor

For the tests of the DP-processor a strategy was used, where first the dependency-graph processor is applied in order to simplify the DP problem. After that, the KBO processor is used to remove some (or all) dependency pairs. If dependency pairs remain, the dependency-graph processor is applied once more, in order to simplify the resulting problem. This strategy is executed repeatedly, up to five times.

Example 8.1.2. The following strategy file was used for the test of the KBO processor, using Minisat as the SAT-solving back-end, where numeric values are encoded as bit-vectors of length 4. For testing purposes, also the statistics and debug output were enabled.

```
1  <Strategy>
2   <DP-Analysis>
3      <Group>
4        <DependencyGraph use_inverse_cap_function="true"
5                         use_strongly_defined_symbols="false" />
6        <Group runs="10">
7          <KBOProcessor word_length="4"
8                        arithmetic_theory="bitvectors"
9                        backend="minisat"
10                       debug="true"
11                       print_statistics="true"/>
12         <DependencyGraph use_inverse_cap_function="true"
13                          use_strongly_defined_symbols="false" />
14       </Group>
15      </Group>
16    </DP-Analysis>
17  </Strategy>
```

Listing 8.2: Strategy for the tests of the KBO processor

[1] http://colo6-c703.uibk.ac.at/ttt2/hz/kbo//index.php, accessed on Sept.22 2011

For the other test cases, the parameters of the KBO processor were modified accordingly.

As for the direct method, 18 test configurations were considered to test the various encodings using different arithmetic theories. Additionally, some tests were made where argument filterings and usable rules were disabled. The next table summarizes the results of all tests. For each test-case we present the number of TRSs in the set of test problems that could be shown to be terminating (YES) or non-terminating (NO). Furthermore, we show the number of problems where termination could not be decided (MAYBE) or the computation exceeded the time limit of 60 seconds (T/O). The table also contains the average execution time of the KBO processor.

Solver	Theory	YES	NO	MAYBE	T/O	Avg. KBO Time	Remark
Yices	Integers	546	88	833	117	1634.28 ms	AF+UR
Yices	Reals	546	88	836	114	1427.88 ms	AF+UR
Yices	BV/2	465	88	922	109	1687.17 ms	AF+UR
Yices	BV/3	544	88	844	108	1887.17 ms	AF+UR
Yices	BV/4	550	88	830	116	2777.71 ms	AF+UR
Yices	BV/5	551	88	795	150	4419.44 ms	AF+UR
Yices	BV/6	549	88	742	205	4895.67 ms	AF+UR
Minisat	BV/4	547	88	822	127	4022.67 ms	AF+UR
Minisat	BV/5	548	88	796	152	5588.74 ms	AF+UR
Minisat	BV/6	546	88	733	217	5466.50 ms	AF+UR
clasp	BV/4	540	88	816	140	3817.51 ms	AF+UR
clasp	BV/5	541	88	800	155	4902.67 ms	AF+UR
clasp	BV/6	540	88	745	211	5746.30 ms	AF+UR
z3	Integers	551	88	837	108	1782.70 ms	AF+UR
z3	Reals	549	88	834	113	1446.42 ms	AF+UR
z3	BV/4	549	88	834	113	2178.62 ms	AF+UR
z3	BV/5	549	88	819	128	3206.10 ms	AF+UR
z3	BV/6	549	88	776	171	4015.60 ms	AF+UR
z3	Integers	321	88	1054	120	2718.38 ms	AF, no UR
z3	Integers	284	88	1108	104	637.07 ms	UR, no AF
z3	Integers	129	88	1265	102	518.77 ms	no AF+UR

Again, the tests were compared to the tests from [ZHM09] where a similar strategy was used. The tests revealed no contradictions between our results and the results from the reference implementation. However, the set of problems for which a solution was found was slightly different for both implementations (considering only the common test-problems). The reason is that the framework of T$_{\text{T}}$T$_2$ (which was used in the tests in [ZHM09]) statically computes the set of usable rules before the KBO processor is used. VMTL also restricts the rules of the input system to a set of usable rules. However, a weaker version of usable rules is used. Therefore, some problems for which a solution was found in the reference implementation yield "MAYBE" in our implementation. On the other hand, there were some problems that could be shown to be terminating in our tests but not in the tests of the reference implementation. Here, the reason

is that VMTL uses a stronger dependency graph processor where sometimes a smaller set of dependency pairs is found.

Since we used a bigger set of test problems, not for all results there were reference solutions available for comparison. For a few random examples, where termination was found, and no reference solution was available, the solutions were checked manually, and found to be correct.

The best results were archived with the z3 SMT-solver using the Integers theory. Using this configuration, 551 TRSs could be proven to be terminating with an average execution-time of the KBO processor of 1782.70 ms. For this test configuration, 108 timeouts occurred during the tests, which is also the lowest number when compared to the number of timeouts of the other configurations.

The tests also showed, that the runtime of the DP processor is much higher than that of the direct method. The main reason is that argument filterings are used, which make the SAT encoding quite big because many properties that could be determined statically in the absence of argument filterings, now have to be encoded as part of the SAT problem (e.g. equality of terms - if argument filterings are used, two terms may be equal under certain argument filterings, while they differ if no argument filtering is used). Disabling argument filterings greatly decreases the runtime of the processor, but also reduces the power of the processor (only 285 systems out of 1584 could be shown to be terminating when argument filterings are disabled, as compared to 551 out of 1584 when argument filterings were enabled).

8.2 Root-Labeling

Due to the huge size of the systems that the root-labeling operations yields in many cases, empirical testing is not feasible for this technique. Instead, root-labeling was tested using systematic tests with various input systems, that should cover all special cases (and all execution paths) that can occur.

The tests revealed a problem with the direct-method version of root-labeling. At first, flat-context stability (resp. flat μ-context stability) was not demanded. However, without flat-context stability, the root-labeling can yield empty systems that are trivially terminating, while the original system is not terminating, making the method unsound.

Example 8.2.1. Let $\mathcal{R}$ consist of the following two rules:

$$\mathsf{a} \rightarrow \mathsf{b} \qquad\qquad \mathsf{b} \rightarrow \mathsf{a}$$

Note, that $\mathcal{R}$ is not flat-context stable. The flat-context closure $\mathcal{FC}_{\mathcal{F}}(\mathcal{R})$ is the empty set and is terminating, while $\mathcal{R}$ is not terminating.

The VMTL implementation of root-labeling checks if flat-context stability (resp. flat μ-context stability) is satisfied, when the method or processor is started. If it is not satisfied then the unchanged input-system is returned.

One test that could be automated, is checking if the number of computed rules is correct. To this end, the number of computed rules was checked against the expected number of rules determined by the formulas presented here.

Let $\mathcal{R}$ be a TRS and let $\mathcal{FC}_{\mathcal{F}}(\mathcal{R})_{\mathtt{rl}}$ be the root-labeled version of $\mathcal{R}$. The following formula gives the number of rules that are contained in $\mathcal{FC}_{\mathcal{F}}(\mathcal{R})_{\mathtt{rl}}$:

$$|\mathcal{FC}_{\mathcal{F}}(\mathcal{R})_{\mathtt{rl}}| = \sum_{l \to r \in \mathcal{R}_p} |\mathcal{F}|^{|\mathtt{Vars}(l)|} + \sum_{l \to r \in \mathcal{R}_a} \left(\sum_{f \in \mathcal{F}} \mathtt{ar}(f) \cdot |\mathcal{F}|^{|\mathtt{Vars}(l)|+\mathtt{ar}(f)-1} \right)$$

Here, $\mathcal{R}_p$ and $\mathcal{R}_a$ denote the root-preserving and root-altering rules of $\mathcal{R}$ as defined in Chapter 6.

For the DP processor, suppose $(\mathcal{P}, \mathcal{R})$ is a DP problem. The root-labeled version of this DP problem is $\mathcal{FC}(\mathcal{P}, \mathcal{R})_{\mathtt{rl}}$. The number of dependency pairs in this problem is given by

$$\sum_{l \to r \in \mathcal{P}} (|\mathcal{F}^{\#}| + 1)^{|\mathtt{Vars}(l)|}$$

and the number of rules is given by

$$\sum_{l \to r \in \mathcal{R}_p} (|\mathcal{F}^{\#}| + 1)^{|\mathtt{Vars}(l)|} + \sum_{l \to r \in \mathcal{R}_a} \left(\sum_{f \in \mathcal{F} \cup \{\Delta\}} \mathtt{ar}(f) \cdot (|\mathcal{F}^{\#}| + 1)^{|\mathtt{Vars}(l)|+\mathtt{ar}(f)-1} \right)$$

For context-sensitive TRSs or CS-DP problems, the presented formulas work as well. However, the term $\mathtt{ar}(f)$ in the base of the formulas has to be replaced by $|\mu(f)|$.

The number of generated dependency pairs and rules was checked against the expected, calculated numbers for the same 1584 problems that were also used when testing KBO. The tests did not reveal any problems. The number of calculated rules and dependency pairs always matched the expected numbers. There was one incident, however, where the formula failed, because one rule was defined twice in the rule set. Here, the set of calculated formulas was correct but the formula presented above fails because the rule should not be considered twice.

As the formulas indicate, the number of rules and dependency pairs can explode even for very small input TRSs. Therefore, it is a good idea to restrict the size of the generated systems by using the provided parameters of the direct method, respectively the DP processor. Furthermore, the practical tests showed that the root-labeling operation either succeeds rather fast (in at most a few seconds) or times out completely for most tested problems. Therefore, when using unrestricted root-labeling in a strategy, it is a good idea to use the root-labeling processor with a low timeout to cut off the cases that would time out anyway due to the exponential blow-up. Furthermore, since the resulting system can get quite huge, this technique should be used at some late point in the strategy, as a kind of last-resort attempt, after all attempts on the original system have failed.

8.3 A New Default Strategy

In most cases, it is a good idea to specify a strategy that is tailored specifically for the TRS that the user wants to examine. However, sometimes it is convenient to have a good general-purpose strategy available that is used if no specific strategy is provided. Another reason, why a good default strategy is required is for the use in competitions. In this section, we present the results for a new default strategy for VMTL. For all tests in this section, a subset of the TRSs in version 8 of the termination problem database was used. The subset we used, contains 92 CS-TRSs, 6 CTRSs and 2762 standard TRSs.

VMTL – Original Version

Here, we present the results for the default strategy of the original VMTL version (VMTL 1.4) where none of the modifications described in this thesis are yet implemented.

This version of VMTL uses a fixed (hard-coded) preprocessing, where the Matrix-Interpretations technique is used to remove some rules. In the default strategy of this version of VMTL, the preprocessing may take up to three seconds and is run up to three times.

After that, the DP-Analysis is started. The first processor that is applied is the Dependency Graph processor. Then, the following processors are used in the given order:

- Subterm Criterion
- Polynomial Ordering (no negative constants, coefficient range 4)
- Matrix Interpretation (dimension 2, coefficient range 3)
- Matrix Interpretation (dimension 3, coefficient range 3)
- Polynomial Ordering (allow negative constants (range 2), coefficient range 8)
- RPOS-Reduction Pair Processor
- SizeChangePrinciple

If at this point, there are any open DP problems left, the following two attempts are made in parallel:

- Forward Narrowing
- Backward Instantiation
- Forward Instantiation
- Propagation

- Backward Narrowing
- Backward Instantiation
- Forward Instantiation
- Propagation

The old default strategy can be looked up in the Appendix in Chapter A.1.

The following table summarizes the test results of the original version of VMTL:

	TRS	CS-TRS	CTRS	Total	
YES	745	65	4	814	VMTL 1.4
NO	116	0	0	116	
MAYBE	122	11	0	133	
TIMEOUT	1778	16	2	1796	

VMTL – Extended Version

Now, we present the results of the tests of the extended VMTL version, with the new preprocessing framework and a new default strategy.

The first table presents the results, when the same default strategy as in the original version is used in the extended version of VMTL. The only difference is, that now preprocessing is done via the new preprocessing framework instead of the old, hard-coded preprocessing.

	TRS	CS-TRS	CTRS	Total	
YES	744	65	4	813	VMTL new
NO	117	0	0	117	
MAYBE	120	11	0	131	
TIMEOUT	1780	16	2	1798	

The results are mostly the same. There were six cases, where one version found "YES" or "NO", while the other version returned "TIMEOUT". These differences arise because of indeterminism introduced through the involved SAT solvers. The results changed if the tests were performed at different times.

The new default strategy extends the old default strategy by incorporating the new methods and processors.

For the preprocessing, after using the Matrix Interpretations technique, now also an attempt is made to show termination with the KBO method (using a timeout of two seconds). This way, some systems can already be shown terminating before starting the DP framework.

In the DP-Analysis, the step where the reduction pair processor was used is now extended. In the old default strategy, the RPO processor and the Dependency Graph processor were executed sequentially up to ten times. The new default strategy also uses the KBO processor in this sequence. Now, first the RPO processor is executed. After that, the KBO processor is executed and then the dependency graph processor is executed. As before, the whole sequence may run up to ten times. The interleaving of RPO and KBO should improve the performance of both processors because some problems, that cannot be treated with the RPO processor, may successfully be treated with the KBO processor and vice versa. Both orders are simplification orders. Therefore, they can treat a similar set of TRSs. Still, the sets of treatable TRSs for both processors are not the same, therefore interleaving both techniques improves the power of both processors.

Another extension, made to the old default strategy, is the integration of root-labeling. If at the very end of the proof search, where the old default strategy would just resign and return "MAYBE", now the root-labeling processor is used in order to change the structure of the TRS. After this has been done, the following processors are applied in the given order:

- Subterm Criterion + Dependency Graph
- An interleaving of RPO, KBO and Dependency Graph
- Polynomial Ordering
- Matrix Interpretation

The new default strategy can be looked up in the Appendix in Chapter A.1.

The following table summarizes the results obtained with the new default strategy.

	TRS	CS-TRS	CTRS	Total	
YES	759	67	4	830	VMTL new
NO	117	0	0	117	
MAYBE	38	6	0	44	
TIMEOUT	1847	19	2	1868	

Comparison with Other Tools

Here, we compare the performance of the extended version of VMTL to VMTL 1.4 (as it was used in the previous competitions) and the results for AProVE and T$_{T_2}$ from the previous competitions.

The following table summarizes the results of the new VMTL version as compared to the results from the termination competition 2010.

	TRS	CS-TRS	Total	
YES	154	21	175	VMTL new
NO	13	0	13	
MAYBE	14	2	16	
TIMEOUT	211	4	215	
YES	133	20	153	VMTL 1.4[1]
NO	10	0	10	
MAYBE	30	4	34	
TIMEOUT	219	3	222	
YES	299	23	322	AProVE[1]
NO	35	1	36	
MAYBE	0	0	0	
TIMEOUT	61	3	61	
YES	216	-	216	TTT2[1]
NO	30	-	30	
MAYBE	146	-	146	
TIMEOUT	0	-	0	
YES	190	23	213	muterm[1]
NO	17	1	18	
MAYBE	45	0	45	
TIMEOUT	140	3	419	
YES	171	-	171	cime3[1]
NO	0	-	0	
MAYBE	137	-	137	
TIMEOUT	84	-	84	

The following table summarizes the results of the new VMTL version as compared to the results from the termination competition 2011.

[1] Results from the termination competition 2010

	TRS	CS-TRS	CTRS	Total	
YES	168	18	4	190	VMTL new
NO	7	0	0	7	
MAYBE	11	2	0	11	
TIMEOUT	404	7	2	185	
YES	167	18	4	167	VMTL 1.4[2]
NO	6	0	0	6	
MAYBE	29	4	0	33	
TIMEOUT	191	5	2	198	
YES	294	19	6	319	AProVE[2]
NO	22	2	0	24	
MAYBE	0	0	0	0	
TIMEOUT	55	6	0	61	
YES	226	-	-	226	TTT2[2]
NO	19	-	-	19	
MAYBE	125	-	-	125	
TIMEOUT	0	-	-	0	

Note, that the results in the two tables presented above should not be seen as a direct comparison, since the results were obtained on different machines (the results from the competitions were obtained during the competitions and the results for the new VMTL version was obtained in dedicated tests on a different machine). However, the results give some idea of the expected improvements. The real performance of VMTL will be tested in the next termination competition in 2012.

[2]Results from the termination competition 2011

CHAPTER 9

Conclusions

9.1 Summary

The general goal of this thesis was to improve VMTL's proof power. To this end, various elements have been added to the core framework of VMTL and to the proof methods of VMTL.

One of the goals was to design and integrate the support of direct proof methods as a preprocessing for the VMTL proof search. In Chapter 9.3, we introduced the abstract notion of direct methods in a way similar to DP processors. Direct methods can be used to either *simplify* term rewriting systems or to directly attempt *termination proofs*. Our notion of direct methods also offers a good degree of modularity, since different methods can be used subsequently in arbitrary order or in parallel to constitute the preprocessing part of the proof search.

The language for the specification of the proof-strategy was also redesigned in the process. Formerly, a proprietary language was used. This language has been replaced by a new XML-based specification format. The new format supports the use of direct methods as preprocessing, as well as the specification of the dependency pair analysis. The new format has some advantages. First, the use of XML-Schema definitions (XSD) allows the generation of specific error reports in case that there are syntactical or logical errors in the strategy specification. Furthermore, XML is a very widespread standard. Thus many tools exist which can be used to create or modify XML files according to a given XSD. XML is also a very portable format and parsers can handle different types of encodings.

In Chapter 5 we introduced the newly developed *VMTL Sat Solving Facility* (VSSF). The VSSF offers a generic interface for SAT and SMT solving tasks. It unifies the way in which SAT/SMT solving is used within VMTL and eases the specification of complex formulas, as it allows the specification of formulas in a very natural way. It automatically translates formulas that are given in propositional logic into SAT-equivalent clause-sets that can be treated by SAT solvers. The VSSF also encapsulates the invocation of concrete SAT and SMT tools and the parsing of their output files. The advantage of our SAT solving facility compared to existing projects like SAT4J [BP10] and OpenSAT [ALBR$^+$03] is that the VSSF also allows the specifi-

cation of SMT problems. Furthermore, the VSSF transparently handles all the management of propositional variables and theory variables. Therefore, propositional (and SMT-) variables can be used in a very natural way, just like other Java-Variables. Compared to standalone SAT/SMT generation tools like URSA [Jan10] and BAT [MSV07], the VSSF has the advantage that all specification can be done directly in Java and the user does not have to deal with intermediate problem specification files.

Formulas specified in the VSSF can be pure propositional formulas but may also contain functions and predicates over different theories. The VSSF offers the arithmetic theories of bit-vectors, integer numbers and real numbers. Depending on the solver that is used, occurrences of functions or predicates from certain theories, which are not directly supported by the chosen solver, are translated into propositional logic if such a translation is possible. If such a translation is not possible, then an exception is generated and a different solver has to be used. Currently, bit-vector predicates and functions are translated to propositional logic by means of bit-blasting.

The VSSF supports the DIMACS format as well as the SMT-LIB 2.0 format. For SAT solvers, propositional formulas are translated into SAT equivalent clause sets. This is done either by using the linear-time Tseitin translation (with the Plaisted-Greenbaum extension) or by a simple (exponential) CNF-transformation. The clause sets that are obtained this way can be translated into DIMACS-input code for use with SAT solvers. The DIMACS-output of the SAT solver is then parsed and a model is extracted. If the original formula contained functions or predicates from some theory that was translated into propositional logic, then the interpretation of the associated variables is extracted from the model and is replaced by the real interpretation of the variable. For SMT solvers, the SMT-LIB 2.0 format is supported. This format offers a rich problem specification language. Therefore, no transformations are necessary. The SMT-LIB 2.0 input for the SMT solvers can be created directly from the problem specification. Again, the output is parsed and a model is extracted. The solving process in the VSSF is illustrated in Figure 9.1.

The VSSF is designed to be very extensible and flexible in its use. To this end, many parameters concerning the solving process can be modified. Extensibility is achieved through the use of Java's object oriented mechanisms.

The first practical use of the VSSF was the implementation of the Knuth-Bendix order (KBO) as a direct method and as a DP processor for VMTL. The implementation is based on the SMT encoding described in [ZHM09]. Our implementation is described in detail in Chapter 7. The use of the VSSF for SMT solving tasks makes it possible to choose among different SAT/SMT solvers at runtime without requiring any code changes. We made a few additions to the SMT-encoding from [ZHM09] in order to support context-sensitive rewriting. For the DP processor, some optimization options were implemented. For example, it is possible to deactivate the use of argument filterings or usable rules. This results in smaller SMT problems, trading proof power for better execution time.

Another proof technique that was added as part of this thesis is called semantic labeling ([Zan95]). More specifically a special case thereof called root-labeling ([SM08], [ST10]) has been integrated into VMTL. In Chapter 6 we gave a short introduction to semantic labeling and generalized the technique of semantic labeling to the context-sensitive case. We also developed

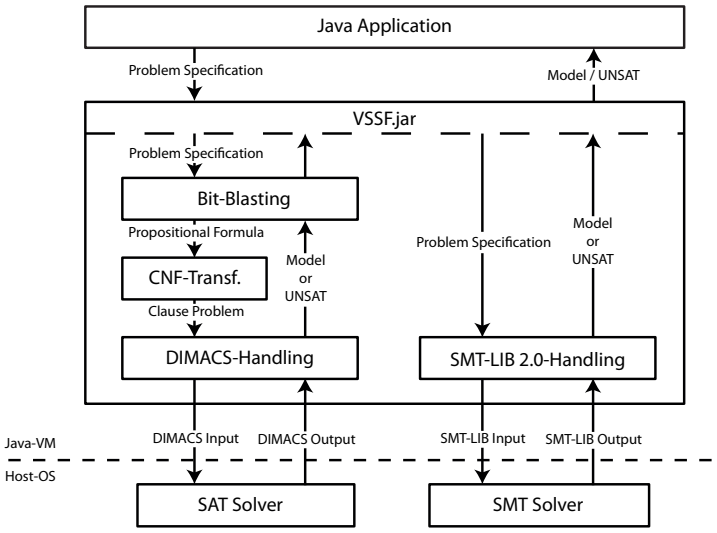


Figure 9.1: Architecture of the VSSF library.

a framework for CS-DP processors for general semantic labeling.

In Sections 6.3 and 6.4 we presented generalizations of the existing root-labeling method and the root-labeling DP processor to context-sensitive versions and proved the soundness of these generalizations. The root-labeling transformation is very well suited for TRSs where the function symbols have a low arity (e.g. in *string rewriting*). For TRSs over function symbols with higher arity, the root-labeling transformation can produce very big systems. Therefore, for some of these systems, root-labeling may not be feasible. In the VMTL implementation of root-labeling, we added parameters by which the user can choose a threshold for the growth of the systems.

Finally, in Chapter 8 we tested the new methods and processors on a number of test-cases from the termination problem database. In this chapter, we also developed a new general-purpose strategy for VMTL that uses preprocessing methods and a DP-analysis. This default-strategy is used, whenever no strategy is provided and is tailored towards use in termination competitions where tools are allowed to run for at most 60 seconds.

9.2 Related Work

Methods for automated termination proofs for term rewriting systems are actively explored. Many termination tools for TRSs have emerged, some of which are: AProVE [GTSKF04], T$_{T_2}$ [KSZM09], TORPA [Zan04], Jambox [EWZ06], Matchbox [Wal04], etc... Like VMTL, most modern termination provers rely on the DP framework [GTSK05] and also support direct proof methods.

Some attempts have been made to generalize the DP-framework to the context-sensitive case.

An early version of the context-sensitive DP framework (CS-DP framework) was presented in [AGL06]. Later, a refined version was published ([AEF+08]), where collapsing dependency pairs are avoided.

The SMT encoding for KBO as a direct method and as a DP processor, on which our implementation is based, is presented in [ZHM09]. For context-free TRSs, when using the direct method, our implementation yields the same results as the implementation from [ZHM09]. The tests of the DP processor did reveal differences in a few cases. The reason for these differences is that the framework used for calculating usable rules and dependency graphs slightly varies between VMTL and TTT2, which was used for the tests in [ZHM09].

The general form of semantic labeling was first presented in [Zan95]. Numerous approaches based on semantic labeling were developed that are suitable for automated termination proving. Examples are, self-labeling [MOZ96], and predictive labeling [HM06]. The root-labeling transformation was first presented in [SM08]. Later, the authors found a problem with one of the claims in this paper and released a refined version of root-labeling which can be found in [ST10]. The refined version of root-labeling was implemented for VMTL as part of this thesis.

Concerning SAT/SMT solving, there exist some projects that provide a SAT API for Java. The two best-known SAT APIs for Java are SAT4J [BP10] and OpenSAT [ALBR+03]. BoolVar [Bai11] is a Java library for encoding pseudo-boolean constraints. Other tools that automate the generation of SAT / SMT specifications are URSA [Jan10] and BAT [MSV07].

9.3 Further Development

The development of a new preprocessing-framework based on direct methods allows the integration of traditional proof methods or transformations other than the DP framework. This is useful since for some problems the DP framework would introduce unnecessary complexity. Furthermore, direct proofs are less convoluted and easier to understand than proofs in the DP framework.

Using the currently implemented direct methods, the proof-power of VMTL could be improved slightly. We are confident that in the future, some stronger preprocessing methods and processors are added that will further improve VMTL's proving power.

A very useful addition to the strategy specification for VMTL would be the possibility for conditional strategies, where parts of the strategy only get executed if certain conditions are met. For example, one could specify that if a system is duplicating then the direct method of KBO is not tried. Another idea would be that certain processors, which are powerful for context-sensitive rewriting, are only tried if the system is context-sensitive.

The VMTL SAT Solving Facility has proven very useful in the implementation of the KBO method and processor. Its flexibility and transparency allow easy experimentation with different solvers and theories in order to find the combination with the best performance for some application. The VSSF is designed with high extensibility in mind. Therefore, it is easy to add new solvers or theories. Planned extensions to the VSSF include support for SMT-LIB 1.6 and new, more efficient CNF-translation techniques like the Jackson-Sharidan approach ([JS04]).

APPENDIX A

Strategy

A.1 Default Strategy

```
[Group,120000,1]{Problem Selection Function=DPNumberProblemEvaluator}(
  [DependencyGraph,0,1]{Use inverse cap function=true,Use strongly defined symbols=false},
  [Group,0,2]{Problem Selection Function=DPNumberProblemEvaluator}(
    [SubtermCriterion,0,1],
    [DependencyGraph,0,1]{Use inverse cap function=true,Use strongly defined symbols=false}
  ),
  [Group,0,10]{Problem Selection Function=DPNumberProblemEvaluator}(
    [PolynomialOrdering,10000,1]{Usable Rules=Improved,Negative Constant Range=0,Coefficient Range
        =4,Polynomial Degree=Linear,Use negative constants=false},
    [DependencyGraph,0,1]{Use inverse cap function=true,Use strongly defined symbols=false}
  ),
  [Group,0,10]{Problem Selection Function=DPNumberProblemEvaluator}(
    [MatrixInterpretations,30000,1]{matrix_dimension=2,coefficient_range=3},
    [DependencyGraph,0,1]{Use inverse cap function=true,Use strongly defined symbols=false}
  ),
  [Group,0,10]{Problem Selection Function=DPNumberProblemEvaluator}(
    [MatrixInterpretations,30000,1]{matrix_dimension=3,coefficient_range=3},
    [DependencyGraph,0,1]{Use inverse cap function=true,Use strongly defined symbols=false}
  ),
  [Group,0,10]{Problem Selection Function=DPNumberProblemEvaluator}(
    [PolynomialOrdering,30000,1]{Usable Rules=Improved,Negative Constant Range=2,Coefficient Range
        =8,Polynomial Degree=Linear,Use negative constants=true},
    [DependencyGraph,0,1]{Use inverse cap function=true,Use strongly defined symbols=false}
  ),
  [Group,0,10]{Problem Selection Function=DPNumberProblemEvaluator}(
    [ReductionPairSAT,60000,1]{Ordering type=rpos,Use usable rules=true,Use nonstrict orderings=
        true,Use argument filterings=true},
    [DependencyGraph,0,1]{Use inverse cap function=true,Use strongly defined symbols=false}
  ),
  [SizeChangePrinciple,30000,1]{Ordering type=embedding,Use usable rules=true},
  [Group,0,1,parallel]{Problem Selection Function=DPNumberProblemEvaluator}(
    [Group,0,3]{Problem Selection Function=DPNumberProblemEvaluator}(
      [Group,120000,1]{Problem Selection Function=DPNumberProblemEvaluator}(
        [ForwardNarrowing,0,100000]{Narrowing normal form lookahead=2,Use original terms
            restriction=true}
      ),
      [BackwardInstantiation,0,1],
      [ForwardInstantiation,0,1],
      [Propagation,0,1]
    ),
    [Group,0,3]{Problem Selection Function=DPNumberProblemEvaluator}(
      [Group,120000,1]{Problem Selection Function=DPNumberProblemEvaluator}(
```

```
        [BackwardsNarrowing,0,100000]{Narrowing normal form lookahead=2,Use original terms
            restriction=true}
      ),
      [BackwardInstantiation,0,1],
      [ForwardInstantiation,0,1],
      [Propagation,0,1]
    )
  )
)
```

Listing A.1: The old default strategy in the old syntax of the strategy language

```
<Strategy>
  <Preprocessing>
    <Group time="5000">
      <MatrixInterpretationsMethod runs="5" matrix_dimension="1" coefficient_range="1" />
      <MatrixInterpretationsMethod runs="5" matrix_dimension="1" coefficient_range="2" />
    </Group>
  </Preprocessing>
  <DP-Analysis>
    <Group>
      <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      <Group runs="2">
        <SubtermCriterion />
        <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      </Group>
      <Group runs="10">
        <PolynomialOrdering time="10000" negative_constant_range="0" use_negative_constants="false"
        coefficient_range="4" usable_rules="Improved" polynomial_degree="Linear" />
        <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      </Group>
      <Group runs="10">
        <MatrixInterpretations time="30000" matrix_dimension="2" coefficient_range="3" />
        <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      </Group>
      <Group runs="10">
        <MatrixInterpretations time="30000" matrix_dimension="3" coefficient_range="3" />
        <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      </Group>
      <Group runs="10">
        <PolynomialOrdering time="30000" negative_constant_range="2" use_negative_constants="true"
        coefficient_range="8" usable_rules="Improved" polynomial_degree="Linear" />
        <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      </Group>
      <Group runs="10">
        <ReductionPairSAT time="60000" ordering_type="rpos" use_usable_rules="true"
        use_argument_filterings="true" use_nonstrict_orderings="true" />
        <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      </Group>
      <SizeChangePrinciple time="30000" ordering_type="embedding" use_usable_rules="true" />
      <Group parallel="true">
        <Group runs="3">
          <Group time="120000">
            <ForwardNarrowing runs="100000" use_original_terms_restriction="true"
            narrowing_normal_form_lookahead="2" />
          </Group>
          <BackwardInstantiation />
          <ForwardInstantiation />
          <Propagation />
        </Group>
        <Group runs="3">
          <Group time="120000">
            <BackwardsNarrowing runs="100000" use_original_terms_restriction="true"
            narrowing_normal_form_lookahead="2" />
          </Group>
          <BackwardInstantiation />
          <ForwardInstantiation />
          <Propagation />
        </Group>
      </Group>
    </Group>
  </DP-Analysis>
```

```
</Strategy>
```

Listing A.2: The old default strategy in XML representation

```
<Strategy>
  <Preprocessing>
    <Group>
      <Group time="5000">
        <MatrixInterpretationsMethod runs="5" matrix_dimension="1" coefficient_range="1" />
        <MatrixInterpretationsMethod runs="5" matrix_dimension="1" coefficient_range="2" />
      </Group>
      <KBO backend="yices" arithmetic_theory="ints" runs="1" time="2000" />
    </Group>
  </Preprocessing>
  <DP-Analysis>
    <Group>
      <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      <Group runs="2">
        <SubtermCriterion />
        <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      </Group>
      <Group runs="10">
        <PolynomialOrdering time="10000" negative_constant_range="0" use_negative_constants="false"
        coefficient_range="4" usable_rules="Improved" polynomial_degree="Linear" />
        <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      </Group>
      <Group runs="10">
        <MatrixInterpretations time="30000" matrix_dimension="2" coefficient_range="3" />
        <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      </Group>
      <Group runs="10">
        <MatrixInterpretations time="30000" matrix_dimension="3" coefficient_range="3" />
        <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      </Group>
      <Group runs="10">
        <PolynomialOrdering time="30000" negative_constant_range="2" use_negative_constants="true"
        coefficient_range="8" usable_rules="Improved" polynomial_degree="Linear" />
        <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      </Group>
      <Group runs="10">
        <Group>
          <ReductionPairSAT time="20000" ordering_type="rpos" use_usable_rules="true"
          use_argument_filterings="true" use_nonstrict_orderings="true" />
          <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
        </Group>
        <Group>
          <KBOProcessor time="20000" backend="yices" arithmetic_theory="ints" />
          <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
        </Group>
      </Group>
      <SizeChangePrinciple time="30000" ordering_type="embedding" use_usable_rules="true" />
      <Group parallel="true">
        <Group runs="3">
          <Group time="120000">
            <ForwardNarrowing runs="100000" use_original_terms_restriction="true"
            narrowing_normal_form_lookahead="2" />
          </Group>
          <BackwardInstantiation />
          <ForwardInstantiation />
          <Propagation />
        </Group>
        <Group runs="3">
          <Group time="120000">
            <BackwardsNarrowing runs="100000" use_original_terms_restriction="true"
            narrowing_normal_form_lookahead="2" />
          </Group>
          <BackwardInstantiation />
          <ForwardInstantiation />
          <Propagation />
        </Group>
      </Group>
      <RootLabelingProcessor max_output_growth="50000" />
```

```
      <Group runs="2">
        <SubtermCriterion />
        <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      </Group>
      <Group runs="10">
        <Group>
          <ReductionPairSAT time="20000" ordering_type="rpos" use_usable_rules="true"
          use_argument_filterings="true" use_nonstrict_orderings="true" />
          <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
        </Group>
        <Group>
          <KBOProcessor time="20000" backend="yices" arithmetic_theory="ints" />
          <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
        </Group>
      </Group>
      <Group runs="10">
        <PolynomialOrdering time="10000" negative_constant_range="0" use_negative_constants="false"
        coefficient_range="4" usable_rules="Improved" polynomial_degree="Linear" />
        <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      </Group>
      <Group runs="10">
        <MatrixInterpretations time="30000" matrix_dimension="2" coefficient_range="3" />
        <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      </Group>
      <Group runs="10">
        <MatrixInterpretations time="30000" matrix_dimension="3" coefficient_range="3" />
        <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      </Group>
      <Group runs="10">
        <PolynomialOrdering time="30000" negative_constant_range="2" use_negative_constants="true"
        coefficient_range="8" usable_rules="Improved" polynomial_degree="Linear" />
        <DependencyGraph use_inverse_cap_function="true" use_strongly_defined_symbols="false" />
      </Group>
    </Group>
  </DP-Analysis>
</Strategy>
```

Listing A.3: The new default strategy in XML representation

A.2 XML Schema Definitions

XSD Template

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Strategy" type="TStrategy" />
  <!-- Type definition for the strategy node -->
  <xsd:complexType name="TStrategy">
    <xsd:sequence>
      <xsd:element name="Preprocessing" type="TPreprocessing" minOccurs="0" maxOccurs="1" />
      <xsd:element name="DP-Analysis" type="TDP-Analysis" minOccurs="0" maxOccurs="1"  />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="TPreprocessing">
    <xsd:choice minOccurs="1" maxOccurs="1">
      <xsd:element name="Group" type="TGroupPreproc" />
      <!-- __GENERATE_DM_ENTRIES__ -->
    </xsd:choice>
  </xsd:complexType>
  <xsd:complexType name="TDP-Analysis">
    <xsd:choice minOccurs="1" maxOccurs="1">
      <xsd:element name="Group" type="TGroupDPAnalysis" />
      <!-- __GENERATE_DP_ENTRIES__ -->
    </xsd:choice>
  </xsd:complexType>
  <!-- abstract type TANode;
       Group and DP-Processor- and DM-nodes extend this -->
  <xsd:complexType name="TANode">
    <xsd:attribute name="time" type="xsd:nonNegativeInteger" default="0" />
    <xsd:attribute name="runs" type="xsd:nonNegativeInteger" default="1" />
  </xsd:complexType>
  <!-- abstract type TAGroup -->
  <xsd:complexType name="TAGroup">
    <xsd:complexContent>
      <xsd:extension base="TANode">
        <xsd:attribute name="parallel" type="xsd:boolean" default="false" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!-- type TGroupPreproc -->
  <xsd:complexType name="TGroupPreproc">
    <xsd:complexContent>
      <xsd:extension base="TAGroup">
        <xsd:choice minOccurs="1" maxOccurs="unbounded">
          <xsd:element name="Group" type="TGroupPreproc" />
          <!-- __GENERATE_DM_ENTRIES__ -->
        </xsd:choice>
        <!-- __GENERATE_DM_EVALUATOR_ATTRIBUTE -->
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!-- type TGroupDPAnalysis -->
  <xsd:complexType name="TGroupDPAnalysis">
    <xsd:complexContent>
      <xsd:extension base="TAGroup">
        <xsd:choice minOccurs="1" maxOccurs="unbounded">
          <xsd:element name="Group" type="TGroupDPAnalysis" />
          <!-- __GENERATE_DP_ENTRIES__ -->
        </xsd:choice>
        <!-- __GENERATE_DP_EVALUATOR_ATTRIBUTE -->
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!-- __GENERATE_DP_TYPES__ -->
  <!-- __GENERATE_DM_TYPES__ -->
</xsd:schema>
```

Listing A.4: XSD-template used by VMTL

Example of Compiled XSD

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Strategy" type="TStrategy" />
  <!-- Type definition for the strategy node -->
  <xsd:complexType name="TStrategy">
    <xsd:sequence>
      <xsd:element name="Preprocessing" type="TPreprocessing" minOccurs="0" maxOccurs="1" />
      <xsd:element name="DP-Analysis" type="TDP-Analysis" minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="TPreprocessing">
    <xsd:choice minOccurs="1" maxOccurs="1">
      <xsd:element name="Group" type="TGroupPreproc" />
      <xsd:element name="TestMethod" type="TTestMethod" />
    </xsd:choice>
  </xsd:complexType>
  <xsd:complexType name="TDP-Analysis">
    <xsd:choice minOccurs="1" maxOccurs="1">
      <xsd:element name="Group" type="TGroupDPAnalysis" />
      <xsd:element name="ForwardNarrowing" type="TForwardNarrowing" />
      <xsd:element name="ForwardInstantiation" type="TForwardInstantiation" />
      <xsd:element name="SizeChangePrinciple" type="TSizeChangePrinciple" />
      <xsd:element name="MatrixInterpretations" type="TMatrixInterpretations" />
      <xsd:element name="BackwardsNarrowing" type="TBackwardsNarrowing" />
      <xsd:element name="BackwardInstantiation" type="TBackwardInstantiation" />
      <xsd:element name="PolynomialOrdering" type="TPolynomialOrdering" />
      <xsd:element name="DependencyGraph" type="TDependencyGraph" />
      <xsd:element name="ReductionPairSAT" type="TReductionPairSAT" />
      <xsd:element name="SubtermCriterion" type="TSubtermCriterion" />
      <xsd:element name="Propagation" type="TPropagation" />
    </xsd:choice>
  </xsd:complexType>
  <!-- abstract type TANode;
       Group and DP-Processor- and DM-nodes extend this -->
  <xsd:complexType name="TANode">
    <xsd:attribute name="time" type="xsd:nonNegativeInteger" default="0" />
    <xsd:attribute name="runs" type="xsd:nonNegativeInteger" default="1" />
  </xsd:complexType>
  <!-- abstract type TAGroup -->
  <xsd:complexType name="TAGroup">
    <xsd:complexContent>
      <xsd:extension base="TANode">
        <xsd:attribute name="parallel" type="xsd:boolean" default="false" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!-- type TGroupPreproc -->
  <xsd:complexType name="TGroupPreproc">
    <xsd:complexContent>
      <xsd:extension base="TAGroup">
        <xsd:choice minOccurs="1" maxOccurs="unbounded">
          <xsd:element name="Group" type="TGroupPreproc" />
          <xsd:element name="TestMethod" type="TTestMethod" />
        </xsd:choice>
        <xsd:attribute name="problem_selection_function_dm" default="SizeEvaluator">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="SizeEvaluator" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!-- type TGroupDPAnalysis -->
  <xsd:complexType name="TGroupDPAnalysis">
    <xsd:complexContent>
      <xsd:extension base="TAGroup">
        <xsd:choice minOccurs="1" maxOccurs="unbounded">
          <xsd:element name="Group" type="TGroupDPAnalysis" />
          <xsd:element name="ForwardNarrowing" type="TForwardNarrowing" />
          <xsd:element name="ForwardInstantiation" type="TForwardInstantiation" />
```

```
72          <xsd:element name="SizeChangePrinciple" type="TSizeChangePrinciple" />
73          <xsd:element name="MatrixInterpretations" type="TMatrixInterpretations" />
74          <xsd:element name="BackwardsNarrowing" type="TBackwardsNarrowing" />
75          <xsd:element name="BackwardInstantiation" type="TBackwardInstantiation" />
76          <xsd:element name="PolynomialOrdering" type="TPolynomialOrdering" />
77          <xsd:element name="DependencyGraph" type="TDependencyGraph" />
78          <xsd:element name="ReductionPairSAT" type="TReductionPairSAT" />
79          <xsd:element name="SubtermCriterion" type="TSubtermCriterion" />
80          <xsd:element name="Propagation" type="TPropagation" />
81        </xsd:choice>
82        <xsd:attribute name="problem_selection_function_dp" default="DPNumberProblemEvaluator">
83          <xsd:simpleType>
84            <xsd:restriction base="xsd:string">
85              <xsd:enumeration value="OverallSizeProblemEvaluator" />
86              <xsd:enumeration value="DPNumberProblemEvaluator" />
87            </xsd:restriction>
88          </xsd:simpleType>
89        </xsd:attribute>
90      </xsd:extension>
91    </xsd:complexContent>
92  </xsd:complexType>
93  <xsd:complexType name="TForwardNarrowing">
94    <xsd:complexContent>
95      <xsd:extension base="TANode">
96        <xsd:attribute name="use_original_terms_restriction" default="true">
97          <xsd:simpleType>
98            <xsd:restriction base="xsd:string">
99              <xsd:enumeration value="true" />
100             <xsd:enumeration value="false" />
101           </xsd:restriction>
102         </xsd:simpleType>
103       </xsd:attribute>
104       <xsd:attribute name="narrowing_normal_form_lookahead" type="xsd:string" default="2" />
105     </xsd:extension>
106   </xsd:complexContent>
107 </xsd:complexType>
108 <xsd:complexType name="TForwardInstantiation">
109   <xsd:complexContent>
110     <xsd:extension base="TANode"></xsd:extension>
111   </xsd:complexContent>
112 </xsd:complexType>
113 <xsd:complexType name="TSizeChangePrinciple">
114   <xsd:complexContent>
115     <xsd:extension base="TANode">
116       <xsd:attribute name="ordering_type" default="embedding">
117         <xsd:simpleType>
118           <xsd:restriction base="xsd:string">
119             <xsd:enumeration value="embedding" />
120             <xsd:enumeration value="lpo" />
121           </xsd:restriction>
122         </xsd:simpleType>
123       </xsd:attribute>
124       <xsd:attribute name="use_usable_rules" default="true">
125         <xsd:simpleType>
126           <xsd:restriction base="xsd:string">
127             <xsd:enumeration value="true" />
128             <xsd:enumeration value="false" />
129           </xsd:restriction>
130         </xsd:simpleType>
131       </xsd:attribute>
132     </xsd:extension>
133   </xsd:complexContent>
134 </xsd:complexType>
135 <xsd:complexType name="TMatrixInterpretations">
136   <xsd:complexContent>
137     <xsd:extension base="TANode">
138       <xsd:attribute name="matrix_dimension" type="xsd:string" default="2" />
139       <xsd:attribute name="coefficient_range" type="xsd:string" default="2" />
140     </xsd:extension>
141   </xsd:complexContent>
142 </xsd:complexType>
143 <xsd:complexType name="TBackwardsNarrowing">
144   <xsd:complexContent>
```

```
      <xsd:extension base="TANode">
        <xsd:attribute name="use_original_terms_restriction" default="true">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="true" />
              <xsd:enumeration value="false" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="narrowing_normal_form_lookahead" type="xsd:string" default="2" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="TBackwardInstantiation">
    <xsd:complexContent>
      <xsd:extension base="TANode"></xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="TPolynomialOrdering">
    <xsd:complexContent>
      <xsd:extension base="TANode">
        <xsd:attribute name="negative_constant_range" type="xsd:string" default="0" />
        <xsd:attribute name="use_negative_constants">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="true" />
              <xsd:enumeration value="false" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="coefficient_range" type="xsd:string" default="4" />
        <xsd:attribute name="usable_rules">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="Standard" />
              <xsd:enumeration value="Improved" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="polynomial_degree">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="Linear" />
              <xsd:enumeration value="Simple Mixed" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="TDependencyGraph">
    <xsd:complexContent>
      <xsd:extension base="TANode">
        <xsd:attribute name="use_inverse_cap_function" default="true">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="true" />
              <xsd:enumeration value="false" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="use_strongly_defined_symbols" default="false">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="true" />
              <xsd:enumeration value="false" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="TReductionPairSAT">
```

```
    <xsd:complexContent>
      <xsd:extension base="TANode">
        <xsd:attribute name="ordering_type" default="rpos">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="rpos" />
              <xsd:enumeration value="lpo" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="use_usable_rules" default="true">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="true" />
              <xsd:enumeration value="false" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="use_argument_filterings" default="true">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="true" />
              <xsd:enumeration value="false" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="use_nonstrict_orderings" default="true">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="true" />
              <xsd:enumeration value="false" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="TSubtermCriterion">
    <xsd:complexContent>
      <xsd:extension base="TANode"></xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="TPropagation">
    <xsd:complexContent>
      <xsd:extension base="TANode"></xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="TTestMethod">
    <xsd:complexContent>
      <xsd:extension base="TANode">
        <xsd:attribute name="result" default="subset">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="exception" />
              <xsd:enumeration value="subset" />
              <xsd:enumeration value="termination" />
              <xsd:enumeration value="non-termination" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="name" type="xsd:string" default="defaultname" />
        <xsd:attribute name="runtime" type="xsd:string" default="1000" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

Listing A.5: VMTL-compiled XSD

APPENDIX B

Example Outputs

B.1 KBO

The Direct Method

In this section, we present an example output of the KBO method. The output contains only the default output (no statistics and no debug information).

Example B.1.1. Consider the following TRS (`TRS_SK90_2.31.trs` from the termination database):

$$\begin{array}{rcl} \mathsf{not}(\mathsf{true}) & \to & \mathsf{false} \\ \mathsf{odd}(0) & \to & \mathsf{false} \\ +(\mathsf{x},0) & \to & \mathsf{x} \\ +(\mathsf{s}(\mathsf{x}),\mathsf{y}) & \to & \mathsf{s}(+(\mathsf{x},\mathsf{y})) \end{array} \qquad \begin{array}{rcl} \mathsf{not}(\mathsf{false}) & \to & \mathsf{true} \\ \mathsf{odd}(\mathsf{s}(\mathsf{x})) & \to & \mathsf{not}(\mathsf{odd}(\mathsf{x})) \\ +(\mathsf{x},\mathsf{s}(\mathsf{y})) & \to & \mathsf{s}(+(\mathsf{x},\mathsf{y})) \end{array}$$

The KBO method can be used to show termination of this TRS. The output created by the KBO method is included on the next page.

In the output, the symbol weight for each function symbol is presented in a table.

The input-system: KBO

Term Rewriting System

Rules

not(true)	→	false	not(false)	→	true
odd(0)	→	false	odd(s(x))	→	not(odd(x))
+(x, 0)	→	x	+(x, s(y))	→	s(+(x, y))
+(s(x), y)	→	s(+(x, y))			

Original Signature

Termination of terms over the following signature is verified: not, 0, s, +, true, false, odd

Strategy

The system can be oriented by a KBO with the following parameters:

Weight function

$w_0 = 1$

Function Symbol	w(·)
not	1
0	2
s	2
false	3
true	2
+	1
odd	1

Quasi precedence

false < s < + < true < 0 < odd < not

The (CS-)DP Processor

In this section, we present an example output of the KBO processor. The output contains only the default output (no statistics and no debug information).

Example B.1.2. Consider the following DP problem that is created by VMTL's dependency graph processor when working on example `TRS_SK90_2.31.trs` from the termination database:

Dependency Pairs:

$$+^{\#}(\mathsf{s}(\mathsf{x}),\mathsf{y}) \quad \to \quad +^{\#}(\mathsf{x},\mathsf{y}) \qquad\qquad +^{\#}(\mathsf{x},\mathsf{s}(\mathsf{y})) \quad \to \quad +^{\#}(\mathsf{x},\mathsf{y})$$

Rules:

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{not}(\mathsf{true}) & \to & \mathsf{false} & \mathsf{not}(\mathsf{false}) & \to & \mathsf{true} \\
\mathsf{odd}(\mathsf{0}) & \to & \mathsf{false} & \mathsf{odd}(\mathsf{s}(\mathsf{x})) & \to & \mathsf{not}(\mathsf{odd}(\mathsf{x})) \\
+(\mathsf{x},\mathsf{0}) & \to & \mathsf{x} & +(\mathsf{x},\mathsf{s}(\mathsf{y})) & \to & \mathsf{s}(+(\mathsf{x},\mathsf{y})) \\
+(\mathsf{s}(\mathsf{x}),\mathsf{y}) & \to & \mathsf{s}(+(\mathsf{x},\mathsf{y})) & & &
\end{array}$$

The KBO processor shows finiteness of this problem by orienting the dependency pair strictly and orienting all usable rules (wrt. the encoded argument filtering) weakly. The output of the KBO processor is included on the following two pages.

In the output, the symbol weight and argument filtering for each function symbol is presented in a table. For the argument filtering, the table contains either a single integer or a (possibly empty) list of integers.

Problem 2: KBOProcessor

Dependency Pair Problem

Dependency Pairs

$+^{\#}(x, s(y)) \rightarrow +^{\#}(x, y)$ $+^{\#}(s(x), y) \rightarrow +^{\#}(x, y)$

Rewrite Rules

$not(true) \rightarrow false$ $not(false) \rightarrow true$

$odd(0) \rightarrow false$ $odd(s(x)) \rightarrow not(odd(x))$

$+(x, 0) \rightarrow x$ $+(x, s(y)) \rightarrow s(+(x, y))$

$+(s(x), y) \rightarrow s(+(x, y))$

Original Signature

Termination of terms over the following signature is verified: not, 0, s, +, true, false, odd

Strategy

The DP-problem after applying the argument filtering π

Dependency Pairs

$+^{\#}(x, s(y)) \rightarrow +^{\#}(x, y)$ $+^{\#}(s(x), y) \rightarrow +^{\#}(x, y)$

Usable rules

There are no usable rules!

Weight Function and Argument Filtering

$w_0 = 2.0$

Function Symbol	π(·)	w(·)
s	[1]	1.0
+#	[1, 2]	0.0

Precedence

s < false < + < true < 0 < +# < not < odd

Strictly Orientable DPs

The following Dependency Pairs could be oriented strictly and can thus be removed

$+^{\#}(x, s(y)) \rightarrow +^{\#}(x, y)$ $+^{\#}(s(x), y) \rightarrow +^{\#}(x, y)$

B.2 Root-Labeling

The Direct Method

Example B.2.1. Consider the following context-sensitive TRS:

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

with the replacement map μ where $\mu(a) = \mu(b) = \emptyset$ and $\mu(f) = \{2, 3\}$.

The output of the direct method implementing root-labeling, applied on this example, is given on the next page.

The input-system: RootLabeling

Term Rewriting System

Rules

$f(a, b, x) \rightarrow f(x, x, x)$ $\qquad$ $a \rightarrow b$

Original Signature

Termination of terms over the following signature is verified: f, b, a

Strategy

Context-sensitive strategy:
$\mu(b) = \mu(a) = \emptyset$
$\mu(f) = \{2, 3\}$

The system was transformed into the following root-labeled system:

$f_{(f,a,a)}(X0, a, X2) \rightarrow f_{(f,b,a)}(X0, b, X2)$ $\qquad$ $f_{(f,a,f)}(X0, a, X2) \rightarrow f_{(f,b,f)}(X0, b, X2)$

$f_{(a,a,a)}(X0, a, X2) \rightarrow f_{(a,b,a)}(X0, b, X2)$ $\qquad$ $f_{(b,a,a)}(X0, a, X2) \rightarrow f_{(b,b,a)}(X0, b, X2)$

$f_{(a,a,f)}(X0, a, X2) \rightarrow f_{(a,b,f)}(X0, b, X2)$ $\qquad$ $f_{(f,a,b)}(X0, a, X2) \rightarrow f_{(f,b,b)}(X0, b, X2)$

$f_{(b,a,f)}(X0, a, X2) \rightarrow f_{(b,b,f)}(X0, b, X2)$ $\qquad$ $f_{(a,a,b)}(X0, a, X2) \rightarrow f_{(a,b,b)}(X0, b, X2)$

$f_{(b,a,b)}(X0, a, X2) \rightarrow f_{(b,b,b)}(X0, b, X2)$ $\qquad$ $f_{(f,f,a)}(X0, X1, a) \rightarrow f_{(f,f,b)}(X0, X1, b)$

$f_{(a,f,a)}(X0, X1, a) \rightarrow f_{(a,f,b)}(X0, X1, b)$ $\qquad$ $f_{(f,b,a)}(X0, X1, a) \rightarrow f_{(f,b,b)}(X0, X1, b)$

$f_{(b,f,a)}(X0, X1, a) \rightarrow f_{(b,f,b)}(X0, X1, b)$ $\qquad$ $f_{(a,b,a)}(X0, X1, a) \rightarrow f_{(a,b,b)}(X0, X1, b)$

$f_{(b,b,a)}(X0, X1, a) \rightarrow f_{(b,b,b)}(X0, X1, b)$ $\qquad$ $f_{(f,a,a)}(X0, X1, a) \rightarrow f_{(f,a,b)}(X0, X1, b)$

$f_{(a,a,a)}(X0, X1, a) \rightarrow f_{(a,a,b)}(X0, X1, b)$ $\qquad$ $f_{(b,a,a)}(X0, X1, a) \rightarrow f_{(b,a,b)}(X0, X1, b)$

$f_{(a,b,a)}(a, b, x) \rightarrow f_{(a,a,a)}(x, x, x)$ $\qquad$ $f_{(a,b,b)}(a, b, x) \rightarrow f_{(b,b,b)}(x, x, x)$

$f_{(a,b,f)}(a, b, x) \rightarrow f_{(f,f,f)}(x, x, x)$

Context-sensitive strategy:
$\mu(b) = \mu(a) = \emptyset$
$\mu(f_{(f,f,b)}) = \mu(f_{(a,b,b)}) = \mu(f_{(f,b,b)}) = \mu(f_{(a,a,f)}) = \mu(f_{(a,b,f)}) = \mu(f_{(f,a,a)}) = \mu(f_{(a,a,a)}) = \mu(f_{(b,b,a)}) = \mu(f_{(f,b,f)}) = \mu(f_{(b,a,a)}) = \mu(f_{(a,f,b)})$
$= \mu(f_{(b,f,b)}) = \mu(f_{(f,f,f)}) = \mu(f_{(f,b,a)}) = \mu(f_{(a,a,b)}) = \mu(f_{(f,f,a)}) = \mu(f_{(a,b,a)}) = \mu(f_{(b,b,f)}) = \mu(f_{(f,a,f)}) = \mu(f_{(f,a,b)}) = \mu(f_{(b,a,f)}) = \mu(f_{(b,b,b)})$
$= \mu(f_{(b,f,a)}) = \mu(f_{(a,f,a)}) = \mu(f_{(b,a,b)}) = \{2, 3\}$

Example B.2.2. Consider again the context-sensitive TRS from example B.2. The context-sensitive dependency pair transformation yields the following CS-DP problem:

Dependency Pairs:

$$\mathsf{f}^{\#}(\mathsf{a},\mathsf{b},\mathsf{x}) \quad \to \quad \mathsf{f}^{\#}(\mathsf{x},\mathsf{x},\mathsf{x})$$

Rules:

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

with the replacement map μ where $\mu(a) = \mu(b) = \emptyset$ and $\mu(f) = \mu(f^{\#}) = \{2, 3\}$.

The output of the direct method implementing root-labeling, applied on this example, is given on the next three pages.

Problem 1: RootLabelingProcessor

Dependency Pair Problem

Dependency Pairs

$$f^{\#}(a, b, x) \rightarrow f^{\#}(x, x, x)$$

Rewrite Rules

$$f(a, b, x) \rightarrow f(x, x, x) \qquad a \rightarrow b$$

Original Signature

Termination of terms over the following signature is verified: f, b, a

Strategy

Context-sensitive strategy:
$\mu(T) = \mu(b) = \mu(a) = \mu(a^{\#}) = \varnothing$
$\mu(f) = \mu(f^{\#}) = \{2, 3\}$

The DP problem was transformed into the following root-labeled DP problem:

Dependency Pair Problem

Dependency Pairs

$$f^{\#}_{(d,d,d)}(d_{(a)}(a), d_{(b)}(b), d_{(a)}(x)) \rightarrow f^{\#}_{(d,d,d)}(d_{(a)}(x), d_{(a)}(x), d_{(a)}(x))$$

$$f^{\#}_{(d,d,d)}(d_{(a)}(a), d_{(b)}(b), d_{(f)}(x)) \rightarrow f^{\#}_{(d,d,d)}(d_{(f)}(x), d_{(f)}(x), d_{(f)}(x))$$

$$f^{\#}_{(d,d,d)}(d_{(a)}(a), d_{(b)}(b), d_{(d)}(x)) \rightarrow f^{\#}_{(d,d,d)}(d_{(d)}(x), d_{(d)}(x), d_{(d)}(x))$$

$$f^{\#}_{(d,d,d)}(d_{(a)}(a), d_{(b)}(b), d_{(b)}(x)) \rightarrow f^{\#}_{(d,d,d)}(d_{(b)}(x), d_{(b)}(x), d_{(b)}(x))$$

$$f^{\#}_{(d,d,d)}(d_{(a)}(a), d_{(b)}(b), d_{(f\#)}(x)) \rightarrow f^{\#}_{(d,d,d)}(d_{(f\#)}(x), d_{(f\#)}(x), d_{(f\#)}(x))$$

Rewrite Rules

$$f_{(d,f,a)}(X0, X1, a) \rightarrow f_{(d,f,b)}(X0, X1, b) \qquad f_{(f,f,a)}(X0, X1, a) \rightarrow f_{(f,f,b)}(X0, X1, b)$$

$$f_{(a,a,a)}(X0, a, X2) \rightarrow f_{(a,b,a)}(X0, b, X2) \qquad f_{(f\#,a,f\#)}(X0, a, X2) \rightarrow f_{(f\#,b,f\#)}(X0, b, X2)$$

$$f_{(a,b,a)}(a, b, x) \rightarrow f_{(a,a,a)}(x, x, x) \qquad f_{(b,f\#,a)}(X0, X1, a) \rightarrow f_{(b,f\#,b)}(X0, X1, b)$$

$$f_{(d,a,f\#)}(X0, a, X2) \rightarrow f_{(d,b,f\#)}(X0, b, X2) \qquad f_{(f,a,d)}(X0, a, X2) \rightarrow f_{(f,b,d)}(X0, b, X2)$$

$$f_{(f\#,a,d)}(X0, a, X2) \rightarrow f_{(f\#,b,d)}(X0, b, X2) \qquad f_{(f\#,a,f)}(X0, a, X2) \rightarrow f_{(f\#,b,f)}(X0, b, X2)$$

$$f_{(a,b,d)}(a, b, x) \rightarrow f_{(d,d,d)}(x, x, x) \qquad f_{(f,a,b)}(X0, a, X2) \rightarrow f_{(f,b,b)}(X0, b, X2)$$

$f_{(f\#,d,a)}(X0, X1, a) \to f_{(f\#,d,b)}(X0, X1, b)$
$f_{(f\#,a,a)}(X0, a, X2) \to f_{(f\#,b,a)}(X0, b, X2)$
$f_{(b,a,f\#)}(X0, a, X2) \to f_{(b,b,f\#)}(X0, b, X2)$
$f_{(a,f,a)}(X0, X1, a) \to f_{(a,f,b)}(X0, X1, b)$
$f_{(b,a,b)}(X0, a, X2) \to f_{(b,b,b)}(X0, b, X2)$
$f_{(a,f\#,a)}(X0, X1, a) \to f_{(a,f\#,b)}(X0, X1, b)$
$f_{(a,a,f\#)}(X0, a, X2) \to f_{(a,b,f\#)}(X0, b, X2)$
$f_{(d,a,b)}(X0, a, X2) \to f_{(d,b,b)}(X0, b, X2)$
$f_{(f,d,a)}(X0, X1, a) \to f_{(f,d,b)}(X0, X1, b)$
$f_{(a,b,a)}(X0, X1, a) \to f_{(a,b,b)}(X0, X1, b)$
$f_{(f,a,f\#)}(X0, a, X2) \to f_{(f,b,f\#)}(X0, b, X2)$
$f_{(b,a,d)}(X0, a, X2) \to f_{(b,b,d)}(X0, b, X2)$
$f_{(d,f\#,a)}(X0, X1, a) \to f_{(d,f\#,b)}(X0, X1, b)$
$f_{(b,a,a)}(X0, a, X2) \to f_{(b,b,a)}(X0, b, X2)$
$f_{(d,a,a)}(X0, X1, a) \to f_{(d,a,b)}(X0, X1, b)$
$f_{(f,f\#,a)}(X0, X1, a) \to f_{(f,f\#,b)}(X0, X1, b)$
$f_{(d,a,f)}(X0, a, X2) \to f_{(d,b,f)}(X0, b, X2)$
$f_{(d,a,d)}(X0, a, X2) \to f_{(d,b,d)}(X0, b, X2)$
$f_{(f\#,f\#,a)}(X0, X1, a) \to f_{(f\#,f\#,b)}(X0, X1, b)$
$f_{(b,d,a)}(X0, X1, a) \to f_{(b,d,b)}(X0, X1, b)$
$f_{(a,b,f\#)}(a, b, x) \to f_{(f\#,f\#,f\#)}(x, x, x)$
$f_{(b,f,a)}(X0, X1, a) \to f_{(b,f,b)}(X0, X1, b)$
$f_{(f\#,f,a)}(X0, X1, a) \to f_{(f\#,f,b)}(X0, X1, b)$
$f_{(f\#,a,b)}(X0, a, X2) \to f_{(f\#,b,b)}(X0, b, X2)$
$f_{(b,a,a)}(X0, X1, a) \to f_{(b,a,b)}(X0, X1, b)$
$f_{(a,a,d)}(X0, a, X2) \to f_{(a,b,d)}(X0, b, X2)$
$f_{(a,b,f)}(a, b, x) \to f_{(f,f,f)}(x, x, x)$
$d_{(a)}(a) \to d_{(b)}(b)$
$f_{(d,a,a)}(X0, a, X2) \to f_{(d,b,a)}(X0, b, X2)$
$f_{(f,b,a)}(X0, X1, a) \to f_{(f,b,b)}(X0, X1, b)$
$f_{(d,d,a)}(X0, X1, a) \to f_{(d,d,b)}(X0, X1, b)$
$f_{(d,b,a)}(X0, X1, a) \to f_{(d,b,b)}(X0, X1, b)$
$f_{(a,a,f)}(X0, a, X2) \to f_{(a,b,f)}(X0, b, X2)$
$f_{(b,b,a)}(X0, X1, a) \to f_{(b,b,b)}(X0, X1, b)$
$f_{(b,a,f)}(X0, a, X2) \to f_{(b,b,f)}(X0, b, X2)$
$f_{(a,a,a)}(X0, X1, a) \to f_{(a,a,b)}(X0, X1, b)$
$f_{(f,a,f)}(X0, a, X2) \to f_{(f,b,f)}(X0, b, X2)$
$f_{(a,b,b)}(a, b, x) \to f_{(b,b,b)}(x, x, x)$
$f_{(f\#,a,a)}(X0, X1, a) \to f_{(f\#,a,b)}(X0, X1, b)$
$f_{(a,a,b)}(X0, a, X2) \to f_{(a,b,b)}(X0, b, X2)$
$f_{(a,d,a)}(X0, X1, a) \to f_{(a,d,b)}(X0, X1, b)$
$f_{(f,a,a)}(X0, a, X2) \to f_{(f,b,a)}(X0, b, X2)$
$f_{(f\#,b,a)}(X0, X1, a) \to f_{(f\#,b,b)}(X0, X1, b)$
$f_{(f,a,a)}(X0, X1, a) \to f_{(f,a,b)}(X0, X1, b)$

Original Signature

Termination of terms over the following signature is verified: $f_{(f,f,b)}$, $f_{(d,a,b)}$, $f_{(a,b,b)}$, $f_{(a,a,f)}$, $f_{(d,f,a)}$, $f_{(f,b,f\#)}$, $d_{(b)}$, $f_{(a,b,f)}$, $f_{(a,d,b)}$, $f_{(f,a,a)}$, $f_{(a,a,a)}$, $f_{(f\#,d,a)}$, $f_{(f,d,b)}$, $f_{(b,b,f\#)}$, $f_{(d,a,f\#)}$, $f_{(f,f\#,a)}$, $f_{(a,f\#,a)}$, $f_{(f,b,d)}$, $f_{(f\#,f,b)}$, $f_{(b,a,f\#)}$, $f_{(a,f,b)}$, $f_{(d,f\#,b)}$, $f_{(f,f,f)}$, $f_{(f\#,f\#,a)}$, $f_{(b,d,b)}$, $f_{(f\#,a,d)}$, $f_{(f,b,a)}$, $f_{(f\#,b,f)}$, $f_{(f\#,f\#,f\#)}$, $f_{(d,b,f)}$, $f_{(a,b,a)}$, $f_{(d,a,a)}$, $f_{(b,b,f)}$, $f_{(a,d,a)}$, $f_{(f,a,f)}$, $d_{(a)}$, $f_{(f\#,b,b)}$, $f_{(f,d,a)}$, $f_{(f,a,b)}$, $f_{(a,a,f\#)}$, $f_{(d,b,b)}$, $f_{(b,a,d)}$, $f_{(a,f\#,b)}$, $f_{(f,a,f\#)}$, $f_{(b,b,b)}$, $f_{(b,f,a)}$, $f_{(d,d,b)}$, $f_{(a,f,a)}$, $f_{(d,a,f)}$, $f_{(a,b,d)}$, $f_{(a,a,d)}$, $f_{(d,a,d)}$, $f_{(f,b,b)}$, $f_{(b,d,a)}$, $f_{(f\#,a,a)}$, $f_{(f\#,f\#,b)}$, $f_{(f\#,b,f\#)}$, $f_{(d,b,f\#)}$, $f_{(f\#,b,a)}$, $f_{(b,b,a)}$, $f_{(f,b,f)}$, $f_{(b,f\#,b)}$, $f_{(d,d,a)}$, $f_{(b,a,a)}$, $f_{(b,f,b)}$, $f_{(f\#,a,f\#)}$, $f_{(f,f,a)}$, b, $f_{(a,a,b)}$, a, $f_{(f\#,a,b)}$, $f_{(f\#,b,d)}$, $f_{(f\#,d,b)}$, $f_{(b,b,d)}$, $f_{(d,b,a)}$, $f_{(f,a,d)}$, $f_{(b,a,f)}$, $f_{(f,f\#,b)}$, $f_{(d,f,b)}$, $f_{(b,f\#,a)}$, $f_{(d,b,d)}$, $f_{(f\#,a,f)}$, $f_{(f\#,f,a)}$, $f_{(d,f\#,a)}$, $f_{(a,b,f\#)}$, $f_{(d,d,d)}$, $f_{(b,a,b)}$

Strategy

Context-sensitive strategy:
$\mu(b) = \mu(a) = \varnothing$
$\mu(d_{(b)}) = \mu(d_{(a)}) = \mu(d_{(f\#)}) = \mu(d_{(d)}) = \mu(d_{(f)}) = \{1\}$
$\mu(f_{(f,f,b)}) = \mu(f_{(d,a,b)}) = \mu(f_{(a,b,b)}) = \mu(f_{(d,f,a)}) = \mu(f_{(a,a,f)}) = \mu(f_{(f,b,f\#)}) = \mu(f_{(a,b,f)}) = \mu(f_{(a,d,b)}) = \mu(f_{(f,a,a)}) = \mu(f_{(a,a,a)}) =$
$\mu(f_{(f\#,d,a)}) = \mu(f_{(d,a,f\#)}) = \mu(f_{(b,b,f\#)}) = \mu(f_{(f,d,b)}) = \mu(f_{(f,f\#,a)}) = \mu(f_{(f,b,d)}) = \mu(f_{(a,f\#,a)}) = \mu(f_{(f\#,f,b)}) = \mu(f_{(b,a,f\#)}) = \mu(f_{(a,f,b)}) =$

$\mu(f_{(f,f,f)}) = \mu(f_{(d,f\#,b)}) = \mu(f_{(f\#,f\#,a)}) = \mu(f_{(b,d,b)}) = \mu(f_{(f\#,a,d)}) = \mu(f_{(f,b,a)}) = \mu(f_{(f\#,b,f)}) = \mu(f_{(f\#,f\#,f\#)}) = \mu(f_{(d,b,f)}) = \mu(f_{(a,b,a)}) = \mu(f_{(d,a,a)}) = \mu(f_{(b,b,f)}) = \mu(f_{(a,d,a)}) = \mu(f_{(f,a,f)}) = \mu(f_{(f,d,a)}) = \mu(f_{(f\#,b,b)}) = \mu(f_{(f,a,b)}) = \mu(f_{(a,a,f\#)}) = \mu(f_{(d,b,b)}) = \mu(f_{(b,a,d)}) = \mu(f_{(a,f\#,b)}) = \mu(f_{(f,a,f\#)}) = \mu(f_{(b,b,b)}) = \mu(f_{(b,f,a)}) = \mu(f_{(d,d,b)}) = \mu(f_{(a,f,a)}) = \mu(f_{(d,a,f)}) = \mu(f_{(a,b,d)}) = \mu(f_{(a,a,d)}) = \mu(f_{(d,a,d)}) = \mu(f_{(f,b,b)}) = \mu(f_{(b,d,a)}) = \mu(f_{(f\#,a,a)}) = \mu(f_{(f\#,f\#,b)}) = \mu(f_{(d,b,f\#)}) = \mu(f_{(f\#,b,f\#)}) = \mu(f_{(f\#,b,a)}) = \mu(f_{(b,b,a)}) = \mu(f_{(f,b,f)}) = \mu(f\#_{(d,d,d)}) = \mu(f_{(b,f\#,b)}) = \mu(f_{(d,d,a)}) = \mu(f_{(b,a,a)}) = \mu(f_{(b,f,b)}) = \mu(f_{(f\#,a,f\#)}) = \mu(f_{(a,a,b)}) = \mu(f_{(f,f,a)}) = \mu(f_{(f\#,a,b)}) = \mu(f_{(f\#,b,d)}) = \mu(f_{(f\#,d,b)}) = \mu(f_{(b,b,d)}) = \mu(f_{(d,b,a)}) = \mu(f_{(f,a,d)}) = \mu(f_{(b,a,f)}) = \mu(f_{(f,f\#,b)}) = \mu(f_{(d,f,b)}) = \mu(f_{(d,b,d)}) = \mu(f_{(b,f\#,a)}) = \mu(f_{(f\#,a,f)}) = \mu(f_{(f\#,f,a)}) = \mu(f_{(a,b,f\#)}) = \mu(f_{(d,f\#,a)}) = \mu(f_{(d,d,d)}) = \mu(f_{(b,a,b)}) = \{2, 3\}$

Bibliography

[AEF+08] B. Alarcón, F. Emmes, C. Fuhs, J. Giesl, R. Gutiérrez, S. Lucas, P. Schneider-Kamp, and R. Thiemann. Improving context-sensitive dependency pairs. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Proc. 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2008), Doha, Qatar, November 22-27, 2008*, volume 5330 of *Lecture Notes in Computer Science*, pages 636–651. Springer-Verlag, 2008.

[AG00] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.

[AGL06] Beatriz Alarcón, Raúl Gutiérrez, and Salvador Lucas. Context-sensitive dependency pairs. In S. Arun-Kumar and Naveen Garg, editors, *Proc. 26th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2006), Kolkata, India, December 13-15, 2006*, volume 4337 of *Lecture Notes in Computer Science*, pages 297–308. Springer-Verlag, 2006.

[ALBR+03] Gilles Audemard, Daniel Le Berre, Olivier Roussel, Ines Lynce, and Joao Marques-Silva. OpenSAT: an open source sat software project. In *Proc. Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03), Santa Margherita Ligure, Italy, 2003*, May 2003. poster.

[Bai11] Olivier Bailleux. BoolVar/PB v1.0, a java library for translating pseudo-boolean constraints into cnf formulae. *CoRR*, abs/1103.3954, 2011.

[BK86] Jan A. Bergstra and Jan Willem Klop. Conditional rewrite rules: Confluence and termination. *J. Comput. Syst. Sci.*, 32(3):323–362, 1986.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[BP10] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli, Sascha Boehme, David Cok, David Deharbe, Bruno Dutertre, Pascal Fontaine, Vijay Ganesh, Alberto Griggio, Jim Grundy, Paul Jackson, Albert Oliveras, Sava Krsti?, Michal Moskal, Leonardo De Moura, Roberto Sebastiani, To David Cok, and Jochen Hoenicke. C.: The smt-lib standard: Version 2.0. Technical report, 2010.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proc. Third Annual ACM Symposium on Theory of Computing, Shaker Heights, Ohio, USA, 1971*, pages 151–158. ACM, 1971.

[DdM06] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at `http://yices.csl.sri.com/tool-paper.pdf`, August 2006.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.

[Dim93] DIMACS challenge. Satisfiability. Suggested format. `http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps`, 1993.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2008), Budapest, Hungary, March 29-April 6, 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer-Verlag, 2008.

[EWZ06] Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. In Ulrich Furbach and Natarajan Shankar, editors, *Proc. Third International Joint Conference on Automated Reasoning (IJCAR 2006), Seattle, WA, USA, August 17-20, 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 574–588. Springer-Verlag, 2006.

[GAO02] Jürgen Giesl, Thomas Arts, and Enno Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *J. Symb. Comput.*, 34(1):21–58, 2002.

[GKNS07] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. clasp : A conflict-driven answer set solver. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *Proc. 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007), Tempe, AZ, USA, May 15-17, 2007*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer-Verlag, 2007.

[GM99] Jürgen Giesl and Aart Middeldorp. Transforming context-sensitive rewrite systems. In Paliath Narendran and Michaël Rusinowitch, editors, *Proc. 10th International Conference on Rewriting Techniques and Applications (RTA-99), Trento, Italy, July 2-4, 1999*, volume 1631 of *Lecture Notes in Computer Science*, pages 271–287. Springer-Verlag, 1999.

[GTSK05] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2004), Montevideo, Uruguay, March 14-18, 2005*, volume 3452 of *Lecture Notes in Computer Science*, pages 301–331. Springer-Verlag, 2005.

[GTSKF04] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated termination proofs with AProVE. In Vincent van Oostrom, editor, *Proc. 15th International Conference on Rewriting Techniques and Applications (RTA 2004), Aachen, Germany, June 3-5, 2004*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220. Springer-Verlag, 2004.

[GTSKF06] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing and improving dependency pairs. *J. Autom. Reasoning*, 37(3):155–203, 2006.

[HM06] Nao Hirokawa and Aart Middeldorp. Predictive labeling. In Frank Pfenning, editor, *Proc. 17th International Conference on Term Rewriting and Applications (RTA 2006), Seattle, WA, USA, August 12-14, 2006*, volume 4098 of *Lecture Notes in Computer Science*, pages 313–327. Springer-Verlag, 2006.

[Jan10] Predrag Janicic. Uniform reduction to SAT. *CoRR*, abs/1012.1255, 2010.

[JS04] Paul Jackson and Daniel Sheridan. Clause form conversions for boolean circuits. In Holger H. Hoos and David G. Mitchell, editors, *Revised Selected Papers 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), Vancouver, BC, Canada, May 10-13, 2004*, volume 3542 of *Lecture Notes in Computer Science*, pages 183–198. Springer-Verlag, 2004.

[Kap84] Stéphane Kaplan. Conditional rewrite rules. *Theor. Comput. Sci.*, 33:175–193, 1984.

[KB70] Donald Knuth and Peter Bendix. Simple word problems in universal algebra. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[KL80] S. Kamin and J.J. Lévy. Two generalizations of the recursive path ordering. *Unpublished Manuscript, University of Illinois, IL, USA*, 1980.

[KNT99] Keiichirou Kusakari, Masaki Nakamura, and Yoshihito Toyama. Argument filtering transformation. In Gopalan Nadathur, editor, *Proc. 1st International Conference on Principles and Practice of Declarative Programming (PPDP'99), Paris, France, September 29 - October 1, 1999*, volume 1702 of *Lecture Notes in Computer Science*, pages 47–61. Springer-Verlag, 1999.

[KO92] Masahito Kurihara and Azuma Ohuchi. Modularity of simple termination of term rewriting systems with shared constructors. *Theor. Comput. Sci.*, 103(2):273–282, 1992.

[KSZM09] Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean termination tool 2. In Ralf Treinen, editor, *Proc. 20th International Conference on Rewriting Techniques and Applications (RTA 2009), Brasília, Brazil, June 29 - July 1, 2009*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304. Springer-Verlag, June 2009.

[LMM05] Salvador Lucas, Claude Marché, and José Meseguer. Operational termination of conditional term rewriting systems. *Inf. Process. Lett.*, 95(4):446–453, 2005.

[Luc95] Salvador Lucas. Fundamentals of context=sensitive rewriting. In Miroslav Bartosek, Jan Staudek, and Jirí Wiedermann, editors, *Proc. 22nd Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM '95), Milovy, Czech Republic, November 23 - December 1, 1995*, volume 1012 of *Lecture Notes in Computer Science*, pages 405–412. Springer-Verlag, 1995.

[Luc98] S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1(1998):1–61, 1998.

[MOZ96] Aart Middeldorp, Hitoshi Ohsaki, and Hans Zantema. Transforming termination by self-labelling. In Michael A. McRobbie and John K. Slaney, editors, *Proc. 13th International Conference on Automated Deduction (CADE-13), New Brunswick, NJ, USA, July 30 - August 3, 1996*, volume 1104 of *Lecture Notes in Computer Science*, pages 373–387. Springer-Verlag, 1996.

[MSV07] Panagiotis Manolios, Sudarshan K. Srinivasan, and Daron Vroon. BAT: The bit-level analysis tool. In Werner Damm and Holger Hermanns, editors, *Proc. 19th International Conference on Computer Aided Verification (CAV 2007), Berlin, Germany, July 3-7, 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 303–306. Springer-Verlag, 2007.

[Ohl02] Enno Ohlebusch. *Advanced topics in term rewriting*. Springer-Verlag, 2002.

[PD07] Knot Pipatsrisawat and Adnan Darwiche. RSat 2.0: Sat solver description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA, 2007.

[PG86] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[RT06] S. Ranise and C. Tinelli. The smt-lib standard: Version 1.2. Technical report, 2006.

[Sch11] Felix Schernhammer. *Applications and Generalizations of Context-Sensitive Term Rewriting*. PhD thesis, TU Wien, Jan. 2011.

[SE05] N. Sörensson and N. Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005:53, 2005.

[SG09] Felix Schernhammer and Bernhard Gramlich. VMTL – a modular termination laboratory. In Ralf Treinen, editor, *Proc. 20th International Conference on Rewriting Techniques and Applications (RTA 2009), Brasília, Brazil, June 29 - July 1, 2009*, volume 5595 of *Lecture Notes in Computer Science*, pages 285–294. Springer-Verlag, Jun 2009.

[SG10] Felix Schernhammer and Bernhard Gramlich. Characterizing and proving operational termination of deterministic conditional term rewriting systems. *J. Log. Algebr. Program.*, 79(7):659–688, 2010.

[SM08] Christian Sternagel and Aart Middeldorp. Root-labeling. In Andrei Voronkov, editor, *Proc. 19th International Conference on Rewriting Techniques and Applications (RTA 2008), Hagenberg, Austria, July 15-17, 2008*, volume 5117 of *Lecture Notes in Computer Science*, pages 336–350. Springer-Verlag, 2008.

[ST10] Christian Sternagel and René Thiemann. Signature extensions preserve termination - an alternative proof via dependency pairs. In Anuj Dawar and Helmut Veith, editors, *Proc. 24th International Workshop on Computer Science Logic (CSL 2010), 19th Annual Conference of the European Association for Computer Science Logic, Brno, Czech Republic, August 23-27, 2010*, volume 6247 of *Lecture Notes in Computer Science*, pages 514–528. Springer-Verlag, 2010.

[Toy87] Yoshihito Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Inf. Process. Lett.*, 25(3):141–143, 1987.

[Tse68] G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.

[Wal04] Johannes Waldmann. Matchbox: A tool for match-bounded string rewriting. In Vincent van Oostrom, editor, *Proc. 15th International Conference on Rewriting Techniques and Applications (RTA 2004), Aachen, Germany, June 3-5, 2004*, volume 3091 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[XHHLB08] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res. (JAIR)*, 32:565–606, 2008.

[Zan95] Hans Zantema. Termination of term rewriting by semantic labelling. *Fundam. Inform.*, 24(1/2):89–105, 1995.

[Zan04] Hans Zantema. Torpa: Termination of rewriting proved automatically. In Vincent van Oostrom, editor, *Proc. 15th International Conference on Rewriting Techniques and Applications (RTA 2004), Aachen, Germany, June 3-5, 2004*, volume 3091 of *Lecture Notes in Computer Science*, pages 95–104. Springer-Verlag, 2004.

[ZHM09] Harald Zankl, Nao Hirokawa, and Aart Middeldorp. Kbo orientability. *J. Autom. Reasoning*, 43(2):173–201, 2009.

Index

$\cdot[\cdot].$, 11
$\Leftarrow$, 18
$\hookrightarrow^*_{\mathcal{R},\mu}$, 15
$\hookrightarrow_{\mathcal{R},\mu}$, 15
$\bot$, 40, 41
$\geq_{\text{kbo}}$, 96
$>_{\text{kbo}}$, 90
$\Box$, 11
$\wedge$, 40, 41
$\oplus$, 40, 41
$\leftrightarrow$, 40, 41
$\rightarrow$, 40, 41
$\vee$, 40, 41
$|\cdot|.$, 10
$[\![\cdot]\!].$, 81
$[\cdot]^{\mathcal{T}}_{\alpha_D}$, 41
$\rightarrow$, 13
$\rightarrow^*$, 13
$\rightarrow_{\mathcal{R}}$, 13
$\rightarrow^*_{\mathcal{R}}$, 13
$\cdot|.$, 11
$\trianglelefteq$, 10
$\trianglelefteq_{\mu'}$, 16
$\trianglelefteq_{\mu}$, 16
$\cdot =^{\pi} \cdot$, 99
$\top$, 40, 41
3-CTRS, 18

$\boldsymbol{\alpha}$
  $\boldsymbol{\mathcal{T}}$-logic, 41
  $\mathcal{F}$-algebra, 70
ADM-SAT$_k(w, w_0)$, 91
admissibility, 89
ADM$^{\pi}_k(\mathcal{F})$, 101
AF$^{\pi}(f)$, 99
`ar`, 9
argument filtering, 93
arity, 9
atomic term, 40

`block`, 77

$\mathcal{C}_\epsilon$-compatibility, 95
$(\mathcal{P}, \mathcal{R}, \mu)$-chain, 17
clause, 47
clause problem, 47
cleaning, 81
closed, 40
closure
  $\mathcal{F}$-contexts, 12
  $\mathcal{T}(\mathcal{F}, X)$-substitutions, 12
completeness
  of CS-DP processors, 17
  of direct methods, 29
  of DP processors, 14
conditional rewriting, 8, 18–20
constant term, 40
context, 11
context-sensitive rewriting, 7–8, 15
CS-DP problem, 17
CS-DP processor, 17
CTRS, 18

DCTRS, 18
defined, 12
dependency pair, 14
  context-sensitive, 17
dependency pair framework, 14

context-sensitive, 16–18
direct method, 29
disjoint theory, 39
DM group node, 31
DM node, 31
DM strategy node, 31
DP group node, 23
DP problem, 14
DP processor, 14
DP processor node, 23
DP strategy node, 23
$DP_c((\mathcal{R},\mu))$, 16
$DP_o((\mathcal{R},\mu))$, 16
$DP(\mathcal{R})$, 14

ϵ, 10

$\mathcal{F}$, 9
$\mathcal{F}$-algebra, 69
factory class, 62
$\mathcal{FC}_{\mathcal{F}}$, 76
$\mathcal{FC}_{\mathcal{F},\mu}$, 79
$\mathcal{FC}_{\mathcal{F}}(\mathcal{P},\mathcal{R},\mu)$, 81
$\mathcal{FC}_{\mathcal{F}}(\mathcal{R})$, 76
$\mathcal{FC}_{\mathcal{F}}(\mathcal{R},\mu)$, 79
$\mathcal{FC}(\mathcal{P},\mathcal{R})$, 77
finiteness
of CS-DP problems, 17, 18
of DP problems, 14
flat μ-contexts, 79
flat context, 76
flat context stability, 77, 79
$\mathcal{F}^{(n)}$, 9
FPos, 11
$\mathtt{FPos}^{\bar{\mu}}(t)$, 15
$\mathtt{FPos}^{\mu}(t)$, 15
Func, 11
$\mathtt{Func}^{\bar{\mu}}(t)$, 15
$\mathtt{Func}^{\mu}(t)$, 15

ground term
$\mathcal{T}$-logic, 40
term rewriting, 11

hidden term, 16
hiding context, 17

instance, 12

KBO, 90
KBO-SAT$_{k,j}(\mathcal{R})$, 92
Knut-Bendix order, 90

lab, 71
labeling function, 71
left-linear, 12
lhs, 12
literal, 47

model
$\mathcal{T}$-logic, 43
$\mathcal{F}$-algebra, 70

$\mathrm{ND}_j^{\pi}(s,t)$, 100

operational termination, 19
oriented CTRS, 18
$\mathrm{ORIENT}_k^{\pi}((\mathcal{P},\mathcal{R}))$, 103

pairwise disjoint theories, 39
parameter, 23
$\pi_{\cdot}$, 71
π, 93
$\mathrm{PO}_j^{\pi}(>)$, 102
Pos, 10
position, 10
$\mathtt{Pos}^{\bar{\mu}}(t)$, 15
$\mathtt{Pos}^{\mu}(t)$, 15
prefix order, 11
$(\overline{\mathcal{P}},\overline{\mathcal{R}},\overline{\mu})$, 74

$\mathrm{QUASI}_j(\succsim)$, 92

redex, 12
reduction order, 13
reduction pair, 93
reduction pair processor, 96
context-sensitive, 98
replacement map, 15
rewrite relation, 13
conditional, 19

context-sensitive, 15
rewrite rule, 12
rhs, 12
$\overline{R}$, 71
$(\overline{\mathcal{R}}, \overline{\mu})$, 72
`root`, 9
root altering, 76
root labeling, 77
root preserving, 76
rule, 12

satisfiable, 43
$\text{SAT}_k(s >_{\text{kbo}} t)$, 92
$\text{SAT}_k(s >^{\pi}_{\text{kbo}} t)$, 101
$\text{SAT}_k(s >^{\pi}_{\text{kbo}}{}' t)$, 101
semantic labeling, 72
σ, 12
signature, 9
SN, *see* trmination13
soundness
of CS-DP processors, 17
of direct methods, 29
of DP processors, 14
strongly conservative, 98
`Sub`, 10
`Subst`, 12
substitution, 12
subsumption, 48
subterm, 10
subterm position, 10
subterm relation
context-sensitive, 16

$\mathcal{T}$, 39
$\mathcal{T}$-formula, 40
$\mathcal{T}$-model, 43
$\mathcal{T}$-term, 40
$\mathcal{T}$-variable assignment, 41
term, 9
term evaluation
$\mathcal{T}$-logic, 41
$\mathcal{F}$-algebra, 70
term rewriting, 5–6, 12–13
term rewriting system, 12
termination, 13
$\mathcal{T}(\mathcal{F}, X)$, 9
theory, 39
bit vectors, 44–45, 52–54
$+$, 44
$-$, 44
$<<_0$, 44
$<<_1$, 44
$=$, 45
$>$, 45
$>>_0$, 44
$>>_1$, 44
$>_s$, 45
`and`, 44
`equiv`, 44
`impl`, 44
`neg`, 44
`or`, 44
`xor`, 44
$\times$, 52
integers, 45–46, 54–55
$*$, 45
$+$, 45
$-$, 45
$=$, 45
$>$, 45
`div`, 45
`mod`, 45
`negative`, 45
`positive`, 45
reals, 46, 55–56
$*$, 46
$+$, 46
$-$, 46
$=$, 46
$>$, 46
$\div$, 46
`div`, 46
`mod`, 46
`negative`, 46
`positive`, 46

μ, see replacement map15
$\mathcal{U}^{\blacktriangleright}(\mathcal{P}, \mathcal{R}, \mu)$, 97

$\mathcal{U}^{\triangleright}(\mathcal{P},\mathcal{R},\mu)$, 97
μ-monotonicity, 97
unravelling, 19
unsatisfiable, 43
μ-Reduction Pair, 97
$\mathcal{U}_{\mathcal{R}}(\mathcal{P},\pi)$, 94
$\mathcal{US}^{\blacktriangleright}(\mathcal{P},\mathcal{R},\mu)$, 97
$\mathcal{US}^{\triangleright}(\mathcal{P},\mathcal{R},\mu)$, 97
usable rules, 94
    context-sensitive, 97
$\mathrm{US}^{\pi}((\mathcal{P},\mathcal{R}))$, 102
$\mathcal{US}_{\mathcal{R}}(x,\pi)$, 94

$\mathtt{val}_{\boldsymbol{\alpha}}^{\mathcal{T}}$, 41
valid, 43
variable assignment
    $\mathcal{T}$-logic, 41
    $\mathcal{F}$-algebra, 70
variable term, 40
$\mathtt{Vars}$, 11
$\mathtt{Vars}^{\bar{\mu}}(t)$, 15
$\mathtt{Vars}^{\mu}(t)$, 15
VMTL, 21
$\mathtt{VPos}$, 11
$\mathtt{VPos}^{\bar{\mu}}(t)$, 15
$\mathtt{VPos}^{\mu}(t)$, 15

$w(\cdot)$, 89
w_0, 89
weight function, 89
well-foundedness, 13
$\mathrm{w}_k^{\pi}(t)$, 100
W_k^t, 92
(w,w_0), 89

X, 9