# Testing the Satisfiability of RPO Constraints

**Diplomarbeit**

am Fachbereich Informatik
der Universität des Saarlandes
von

**Jan-Georg Timm**

Saarbrücken, 1997

Erklärung
Ich erkläre, die vorliegende Arbeit selbständig im Sinne der Diplomprüfungsordnung erstellt und ausschließlich die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Saarbrücken, 23. Oktober 1997

Jan-Georg Timm

Betreuer: Prof. Dr. Harald Ganzinger
Dr. Christoph Weidenbach

To my parents Helga Timm and Georg–Wilhelm Timm


## Acknowledgments

I would like to thank Prof. Dr. Harald Ganzinger, who not only made it possible for me to write my thesis about such an interesting subject, but also agreed to become one of the referees. I also wish to acknowledge Prof. Dr. Jörg Siekmann for becoming the other referee. I am especially indebted to my supervisor Dr. Christoph Weidenbach for the efficiency with which he advised me whenever I asked. In this place thanks also to Ullrich Hustadt who helped me find such a friendly, patient and competent supervisor. Finally, I want to thank Noëmi Preissner and Uwe Brahm for their proofreading.

# Contents

# List of Figures

# List of Tables

# 1

# Introduction

## 1.1 Motivation

In this work, I will present an algorithm to check the satisfiability of RPO constraints and describe the implementation of the introduced algorithm. The recursive path ordering RPO generalizes both the lexicographic path ordering (LPO) and the multiset path ordering by allowing each function symbol to have a lexicographic or multiset status. The RPO is a total reduction quasi–ordering on ground terms.

An important application of ordering constraints are ordered strategies in automated deduction in first order logic, e.g. the superposition calculus of L. Bachmair and H. Ganzinger[BG94] which requires a reduction ordering, total on ground terms. Other works on automated deduction include [KKR90] and [NR92]. In [RN93] A. Rubio and R. Nieuwenhuis introduce an AC–compatible ordering based on RPO.

H. Comon proved that the decidability of LPO constraints is satisfiable [Com90a], J.P. Jouannaud and M. Okada proved the same for RPO constraints [JO91]. In both papers, a satisfiability check algorithm is constructed for proof purposes, but these algorithms are neither easy to implement nor very efficient. A. Rubio and R. Nieuwenhuis introduced an elegant and efficient algorithm for the LPO case with a restricted precedence [RN91, Rub94a]. Unfortunately, no such algorithm was known for RPO constraints. Also, the method of Rubio and Nieuwenhuis cannot be simply adopted for RPO constraints, as the successor function on terms is not total for RPO (not even with the restrictions on the precedence introduced in [RN91]).

Ch. Weidenbach managed to overcome the necessity of a total successor function in his algorithm [Wei94] and, moreover, his algorithm works for unrestricted preferences. In order to allow arbitrary–arity multiset–status function symbols, I modified his algorithm; the result is presented in Chapter 3 and also used in my implementation.

A short comparison of the different approaches in [Com90a] and [JO91] on the one hand and [RN91], [Wei94] and this work on the other hand is given in Chapter 3, Section 3.3.

## 1.2 First Example

Now we will look at a *very small* example to illustrate the problem.
Consider the constraint $C = \{f(x,y) \succ g(x)\}$ with precedence $g > f > 0$ and $\mathrm{Stat}(f) = \mathrm{mul}$, $\mathrm{Stat}(g) = \mathrm{lex}$. Applying our rewrite system yields:

$$\begin{aligned} & f(x,y) \succ g(x) \rightarrow \\ & x \succ g(x) \lor y \succ g(x) \end{aligned}$$

Next, the rewrite system reduces $x \succ g(x)$ to $\perp$. Hence, only the solved problem $y \succ g(x)$ is left. As

$$y \succ g(x) \text{ satisfiable } \Leftrightarrow y \simeq \mathrm{succ}(g(x)) = f(g(x),0) \text{ satisfiable}$$

the algorithm now checks if $y \simeq f(g(x),0)$ is satisfiable. $y \simeq f(g(x),0)$ is solved and thus the constraint is satisfiable, for instance by the solution $\{x = 0, y = f(g(0),0)\}$.

## 1.3 The Implementation

The implementation of the constraint solver was done in about 6400 lines of `C++` code and uses the library `EARL` [WMCK95] with its data structures for symbols, signatures, terms and formulas.

As a first exercise, I implemented the RPO for ground terms using the bottom–up approach suggested by W. Snyder [Sny93]. This program is similar to the simplifier described in Chapter 4, and turned out to be useful to check the solutions computed by the constraint solver.

The proof of the satisfiability of RPO constraints by constructing an efficient algorithm in theory still leaves various problems that have to be solved for the actual implementation. Among these are

- incremental computation of solved problems
- choosing an efficient application order for the rewrite rules

- cycle checks for some rewrite rules
- efficient computation of permutations
- a simplifier to reduce the problem

How these problems and considerations were handled in the implementation is described in Chapter 4.

## 1.4 Conclusion

My thesis is based on the satisfiability check algorithm for RPO constraints presented by Ch. Weidenbach in [Wei94]. However, contrary to his version, mine allows arbitrary–arity multiset–status function symbols. I proved that the presented rewrite system for RPO constraints is complete, correct and terminating. Furthermore, I showed the correctness of the definition of the successor function on terms wrt. $\succ_{\mathrm{rpo}}$.

I implemented the algorithm and developed a method to compute solved problems incrementally. Finally, in order to improve the overall performance of the program, I thought up and integrated a simplifier.

# 2

# Preliminaries

This thesis deals with RPO constraints, which are quantifier-free *first-order logic* formulas over syntactic equations and inequations . Therefore, we will first introduce the standard notations and definitions for first-order logic as far as needed here. Readers familiar with this topic may skip the first section. For a complete introduction see for instance [Fit90].

Since we introduce our algorithm as sets of don't–care non–deterministic rewrite rules, the second section introduces *term rewriting* (see [Der87] for a detailed introduction).

Finally, *orderings on terms* and *ordering constraints* are introduced. Many of the notations and definitions there are consistent with or taken from [Rub94a] and [DJ90].

## 2.1 First-Order Logic

**Definition 2.1.1 (Signature)**
A signature $\Sigma = (\mathcal{X}, \mathcal{F}, \mathcal{P})$ consists of the following disjoint sets:

- $\mathcal{X}$ is the countable infinite set of variable symbols.
- $\mathcal{F}$ is the countable infinite set of function symbols. It is the union of the sets of $n$-place function symbols $\mathcal{F}_n$ $(n \in \mathbb{N}_0)$ and the set of function symbols with arbitrary arity $\mathcal{F}_a$ (called *variadic* function symbols). The 0-place function symbols in $\mathcal{F}_0$ are also called *constants*.

- $\mathcal{P}$ is the finite set of predicate symbols divided into the sets of $n$-place predicate symbols $\mathcal{P}_n$.

We will denote variables by $x, y, z$ (possibly with subscripts), $f, g, h$ will be used for function symbols and $0, a, b, c$ for constants.

**Definition 2.1.2 (Special Symbols)**

- The propositional constants $\top$ and $\bot$ ("true" and "false")
- The logical connectives $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$
- The quantifiers $\forall, \exists$
- The punctuation symbols "(", ")", ", "

**Definition 2.1.3 (Terms)**
The set of *terms* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the smallest set with $\mathcal{X} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ for $f \in \mathcal{F}_n \cup \mathcal{F}_a$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. The set of variables $\mathit{Vars}(t)$ occuring in term $t$ is defined as

$$\begin{aligned} \mathit{Vars}(x) &= \{x\} \\ \mathit{Vars}(a) &= \emptyset \\ \mathit{Vars}(f(t_1, \ldots, t_n)) &= \bigcup_{i=1}^{n} \mathit{Vars}(t_i) \end{aligned}$$

A term without variables is called *ground*, $\mathcal{T}(\mathcal{F})$ is the set of all *ground terms*. The *top symbol* of a term is denoted by the function top:

$$\begin{aligned} \mathrm{top}(x) &= x \\ \mathrm{top}(a) &= a \\ \mathrm{top}(f(t_1, \ldots, t_n)) &= f \end{aligned}$$

The *arity* of a term is defined as

$$\begin{aligned} \mathrm{arity}(x) &= 0 \\ \mathrm{arity}(a) &= 0 \\ \mathrm{arity}(f(t_1, \ldots, t_n)) &= n \end{aligned}$$

The *size* of a term size$(t)$ is defined as

$$\begin{aligned} \mathrm{size}(x) &= 1 \\ \mathrm{size}(a) &= 1 \\ \mathrm{size}(f(t_1, \ldots, t_n)) &= 1 + \sum_{i=1}^{n} \mathrm{size}(t_i) \end{aligned}$$

**Definition 2.1.4 (Position)**
A *position* within a term is represented by a finite sequence of positive integers. The subterm of a term $t$ at position $p$, called *occurrence*, is denoted $t|_p$ and defined as follows:

$$\begin{array}{rcll} t|_p & = & t & \text{for } p = \lambda \\ f(t_1, \dots, t_n)|_p & = & t_i|_q & \text{for } p = i.q \ (1 \leq i \leq n), \\ & & & q \text{ sequence of positive integers.} \end{array}$$

A term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ can be viewed as a tree: the root is labeled with the topsymbol of $t$, leaves are labeled with variables in $\mathcal{X}$ and 0-*ary* function symbols in $\mathcal{F}$ (called *constants*), and internal nodes are labeled with the topsymbols of subterms. The outdegree of a node is the arity of the subterm rooted in this node. (See Figure 2.1 for an example). As we saw in Definition 2.1.4, a position within a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is represented by a sequence of positive integers: the path from the topsymbol (the root of the graph) to the topsymbol of a subterm at that position.

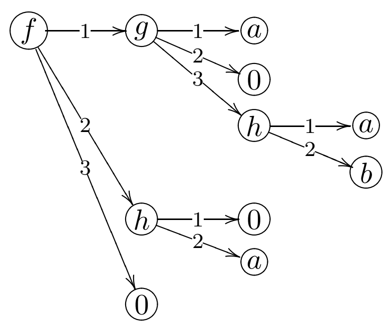



Figure 2.1: Tree representation of term $t = f(g(a, 0, h(a, b)), h(0, a), 0)$

**Definition 2.1.5 (Context)**
A *context* $u[\ ]_p$ is a term with a "hole" at position $p$. Given some term $t$ and some context $u[\ ]_p$ the expression $u[t]_p$ is again a term. A term $t$ with its subterm $t|_p$ replaced by term $s$ is denoted $t[s]_p$.

**Definition 2.1.6 (Substitution)**
A *substitution* is a mapping $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F}, \mathcal{X})$ from the set of Variables $\mathcal{X}$ to the set of terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that the *domain* $\mathrm{Dom}(\sigma) = \{x \in \mathcal{X} \mid x\sigma \neq x\}$ is finite. The application of the substitution $\sigma$ to a term $t$ is denoted $t\sigma$. A substitution is *ground* if its range is $\mathcal{T}(\mathcal{F})$.

Although substitutions are mappings on variables, they are easily extended to all terms.

**Definition 2.1.7**
Let $\sigma$ be a substitution. Then we set:

1. $a\sigma = a$ for a constant $a$;
2. $[f(t_1, \dots, t_n)]\sigma = f(t_1\sigma, \dots, t_n\sigma)$ for an $n$-place or arbitrary arity function symbol $f$.

## 2.2 Term Rewriting

**Definition 2.2.1 (Multisets)**
A *multiset* over a set $Y$ is a function $M : Y \to \mathbb{N}$:

1. $M(y), y \in Y$ is the number of occurrences of the element $y$ in $M$, also called *multiplicity*. $y \in M$ if $M(y) > 0$.
2. $(M_1 \cup M_2)(y) = M_1(y) + M_2(y)$
3. $(M_1 \cap M_2)(y) = \min(M_1(y), M_2(y))$

We use a set-like notation for multisets, i.e. $\{a, a, b\}$ denotes the multiset $M$ with $M(a) = 2$, $M(b) = 1$ and $M(y) = 0$ for $y \notin \{a, b\}$. If the elements of a multiset are also multisets, then we speak of an $n$-fold multiset, where $n$ is the number of nested multisets. Example: $\{\{a, a\}, \{a, b, b\}, \{a, b, b\}\}$ is a 2-fold multiset.

**Definition 2.2.2 (Equation, Rewrite Rule, Term Rewrite System)**
An *equation* is a multiset of terms $\{s, t\}$, denoted $s \simeq t$. A *rewrite rule* is an ordered pair of terms $\langle s, t\rangle$, written $s \to t$. A set of rewrite rules $\mathcal{R}$ over $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is called a *term rewrite system* or simply *rewrite system* .

Let $\mathcal{R}$ be a rewrite system and $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ terms. Then $s$ rewrites into $t$, denoted $s \to_{\mathcal{R}} t$, if $s|_p = l\sigma$ and $t = s[r\sigma]_p$, for some rule $l \to r$ in $\mathcal{R}$, position $p$ in $s$ and substitution $\sigma$.

If $\to$ is a binary relation, then $\leftarrow$ is the inverse, $\leftrightarrow$ the symmetric closure, $\xrightarrow{+}$ the transitive closure and $\xrightarrow{*}$ the reflexive-transitive closure.

**Definition 2.2.3 (Properties of Rewrite Systems)**

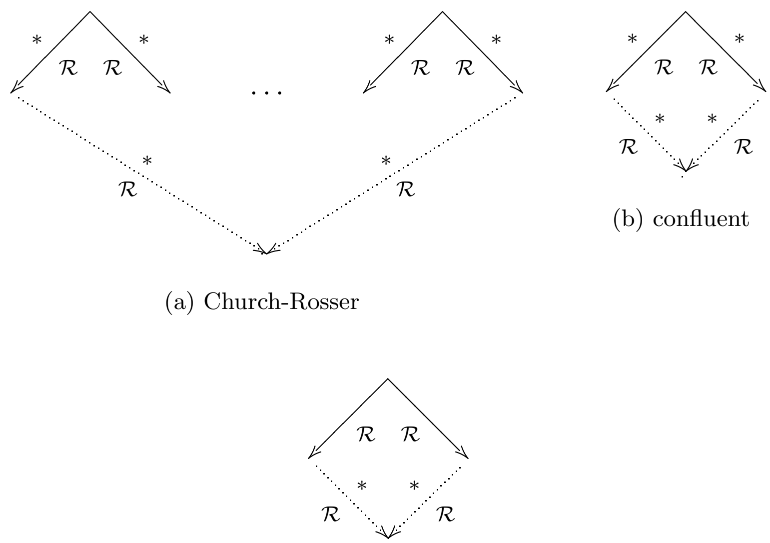
**Termination:** A rewrite system $\mathcal{R}$ is *terminating* if there is no infinite sequence $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \dots$ of terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

**Normal Form** A term $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is in *normal form* or *irreducible* wrt. a rewrite system $\mathcal{R}$ if there is no term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $s \to_{\mathcal{R}} t$. A term $t$ is a normal form of term $s$ wrt. $\mathcal{R}$ if $s \xrightarrow{*}_{\mathcal{R}} t$ and $t$ is irreducible.

**Set of Normal Forms:** If $\mathcal{R}$ is terminating then every term has at least one normal form. Let $\mathcal{R}$ be a terminating rewrite system and $t$ a term. The *set of normal forms* of $t$ wrt. $\mathcal{R}$ is denoted by $\mathrm{snf}_{\mathcal{R}}(t)$.

**Convergence:** A rewrite system $\mathcal{R}$ is convergent if every (ground) term $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ ($s \in \mathcal{T}(\mathcal{F})$) has a unique normal form.

**Church-Rosser:** A rewrite system $\mathcal{R}$ is called Church-Rosser if the reflexive-transitive closure $\xleftrightarrow{*}$ is contained in the joinability relation $\xrightarrow{*}_{\mathcal{R}} \circ \xleftarrow{*}_{\mathcal{R}}$. (See Fig. 2.2(a))



(a) Church-Rosser

(b) confluent

(c) locally confluent

Figure 2.2: Joinability properties of rewrite systems

This is equivalent to the following property:

**Confluence:** A rewrite system $\mathcal{R}$ is called confluent if the relation $\xleftarrow{*}_{\mathcal{R}} \circ \xrightarrow{*}_{\mathcal{R}}$ is contained in the joinability relation $\xrightarrow{*}_{\mathcal{R}} \circ \xleftarrow{*}_{\mathcal{R}}$. (See Fig. 2.2(b)).

Terminating confluent rewrite systems are convergent.

**Local Confluence:** A rewrite system $\mathcal{R}$ is called locally confluent if any local divergence $\leftarrow_{\mathcal{R}} \circ \rightarrow_{\mathcal{R}}$ is contained in the joinability relation $\xrightarrow{*}_{\mathcal{R}} \circ \xleftarrow{*}_{\mathcal{R}}$. (See Fig. 2.2(c)).

A terminating rewrite system is confluent iff it is locally confluent (*Diamond Lemma*)

## 2.3 Orderings on Terms

Now we introduce some definitions and classifications for orderings on terms.

**Definition 2.3.1 (Properties of Orderings)**

- A (strict partial) ordering $\succ$ is a transitive and irreflexive binary relation.
- An ordering $\succ$ is called *monotonic* or *closed under context application* if

$$s \succ t \Rightarrow u[s]_p \succ u[t]_p \qquad \forall s,t \in \mathcal{T}(\mathcal{F},\mathcal{X}) \text{ and for all positions } p \text{ and contexts } u$$

- An ordering $\succ$ is called *stable* under substitution if

$$s \succ t \Rightarrow s\sigma \succ t\sigma \qquad \forall s,t \in \mathcal{T}(\mathcal{F},\mathcal{X}) \text{ and substitutions } \sigma$$

- An ordering $\succ$ is called *well-founded* if there is no infinite decreasing sequence

$$t_1 \succ t_2 \succ t_3 \succ \dots$$

- An ordering $\succ$ possesses the *subterm property* if

$$t[s]_p \succ s \qquad \forall s,t \in \mathcal{T}(\mathcal{F},\mathcal{X}) \text{ and } p \neq \lambda$$

- An ordering $\succ$ possesses the *deletion property* if

$$s = f(t_1,\dots,t_n) \succ f(t_1,\dots t_{i-1},t_{i+1},\dots,t_n) \quad \forall s \in \mathcal{T}(\mathcal{F},\mathcal{X}), \; f \text{ has arbitrary arity}$$

**Definition 2.3.2 (Rewrite Ordering)**
An ordering on terms is called *rewrite ordering*, if it is monotonic and stable under substitutions, i.e.:

$$s \succ t \Rightarrow u[s\sigma]_p \succ u[t\sigma]_p \qquad \forall s,t \in \mathcal{T}(\mathcal{F},\mathcal{X}), \text{ positions } p, \text{ contexts } u \text{ and substitutions } \sigma$$

**Definition 2.3.3 (Reduction Ordering)**
A *reduction ordering* on a set of terms $\mathcal{T}(\mathcal{F},\mathcal{X})$ is any *well-founded* rewrite ordering on $\mathcal{T}(\mathcal{F},\mathcal{X})$.

**Definition 2.3.4 (Simplification Ordering)**
A rewrite ordering possessing the deletion property and the subterm property is a *simplification ordering*.

**Theorem 2.3.5**
*For a finite set of function symbols $\mathcal{F}$ any simplification ordering on $\mathcal{T}(\mathcal{F},\mathcal{X})$ is well-founded and therefore also a reduction ordering.*

For the construction of path orderings the following definitions about lexicographic and multiset extensions of orderings and congruences are needed.

**Definition 2.3.6 (Lexicographic Extension of $\succ$)**
Let $\succ$ be an ordering and $\simeq$ a congruence relation. The lexicographic (left to right) extension $\succ^{\text{lex}}$ of $\succ$ wrt. $\simeq$ for $n$-tuples is defined as:

$$\langle s_1,\dots,s_n\rangle \succ^{\text{lex}} \langle t_1,\dots,t_n\rangle \qquad \text{iff}$$
$$s_1 \simeq t_1,\dots,s_{k-1} \simeq t_{k-1} \text{ and } s_k \succ t_k \text{ for some } 1 \le k \le n$$

The ordering $\succ^{\text{lex}}$ is well-founded if $\succ$ is well-founded.

**Definition 2.3.7 (Multiset Extension of $\simeq$)**
Let $\simeq$ be a congruence relation. Its multiset extension, denoted $\simeq^{\text{mul}}$ is defined as:

$$\{s_1,\dots,s_m\} \simeq^{\text{mul}} \{t_1,\dots,t_n\} \qquad \text{iff}$$
$$m = n \text{ and } n > 0 \Rightarrow \exists\, t_j : s_1 \simeq t_j \text{ and}$$
$$\{s_1,\dots,s_m\} \setminus \{s_1\} \simeq^{\text{mul}} \{t_1,\dots,t_n\} \setminus \{t_j\}$$

**Definition 2.3.8 (Multiset Extension of $\succ$)**
Let $\succ$ be an ordering and $\simeq$ a congruence relation. Its extension wrt. $\simeq$ to finite multisets, denoted $\succ^{\text{mul}}$, is defined as:
$M = \{s_1,\dots,s_m\} \succ^{\text{mul}} \{t_1,\dots,t_n\} = N$ if

- $M \neq \varnothing$ and $N = \varnothing$ or
- $s_i \simeq t_j$ and $M \setminus \{s_i\} \succ^{\text{mul}} N \setminus \{t_j\}$, for some $1 \le i \le m$ and $1 \le j \le n$ or
- $s_i \succ t_{j_1} \wedge \dots \wedge s_i \succ t_{j_k}$ and
  $M \setminus \{s_i\} \succ^{\text{mul}} N \setminus \{t_{j_1},\dots,t_{j_k}\}$ or $M \setminus \{s_i\} \simeq^{\text{mul}} N \setminus \{t_{j_1},\dots,t_{j_k}\})$
  for some $1 \le i \le m$ and $1 \le j_1 < \dots < j_k \le n$ $(1 \le k \le n)$.

The ordering $\succ^{\text{mul}}$ is well-founded if $\succ$ is well-founded.

Now we will introduce two examples of simplification orderings: the *lexicographic path ordering*, short *LPO*, and the *recursive path ordering with status*, short *RPO*.

**Definition 2.3.9 (Precedence)**
A well-founded, total ordering $>_{\mathcal{F}}$ on the set of function symbols $\mathcal{F}$ is called *precedence* (and often is just denoted $>$).

**Definition 2.3.10 (LPO)**
The *lexicographic path ordering* (LPO) generated by a precedence $>_{\mathcal{F}}$ on the set of function symbols $\mathcal{F}$, denoted by $\succ_{\text{lpo}}^{\mathcal{F}}$ (or simply $\succ_{\text{lpo}}$), is defined as:
$s = f(s_1, \dots, s_n) \succ_{\text{lpo}} g(t_1, \dots, t_n) = t$ if

1. $f >_{\mathcal{F}} g$ and $s \succ_{\text{lpo}} t_j$, for all $1 \leq j \leq n$ or
2. $s_i \succeq_{\text{lpo}} t$, for some $1 \leq i \leq n$ or
3. $f = g$, $\langle s_1, \dots, s_n \rangle \succ_{\text{lpo}}^{\text{lex}} \langle t_1, \dots, t_n \rangle$ and $s \succ_{\text{lpo}} t_j$, for all $1 \leq j \leq n$

**Proposition 2.3.1**
LPO is a rewrite ordering and a simplification ordering. Moreover, if $\mathcal{F}$ is finite, LPO is a reduction ordering.

**Definition 2.3.11 ($\simeq_{\text{rpo}}$)**
For Definition 2.3.12 we first need to define $\simeq_{\text{rpo}}$: Let $>_{\mathcal{F}}$ be a precedence on a set of function symbols $\mathcal{F}$. We denote by $\text{Stat}(f)$ the *status* of a function symbol $f$ which can be either lex (lexicographic status) or mul (multiset status). Then
$f(s_1, \dots, s_m) \simeq_{\text{rpo}} g(t_1, \dots, t_n)$ if $f = g$, $m = n$ and

1. $\text{Stat}(f) = \text{lex}$ and $s_i \simeq_{\text{rpo}} t_i$, for all $1 \leq i \leq m$ or
2. $\text{Stat}(f) = \text{mul}$ and $\{s_1, \dots, s_m\} \simeq_{\text{rpo}}^{\text{mul}} \{t_1, \dots, t_n\}$

**Definition 2.3.12 (RPO)**
The *recursive path ordering with status* (RPO) generated by a precedence $>_{\mathcal{F}}$ on a set of function symbols $\mathcal{F}$ with status (where the function symbols with lexicographic status must have fixed arity), denoted by $\succ_{\text{rpo}}^{\mathcal{F}}$ (or simply $\succ_{\text{rpo}}$), is defined as:
$s = f(s_1, \dots, s_m) \succ_{\text{rpo}} g(t_1, \dots, t_n) = t$ if

1. $f >_{\mathcal{F}} g$ and $s \succ_{\text{rpo}} t_j$, for all $1 \leq j \leq n$ or
2. $g >_{\mathcal{F}} f$ and $s_i \succeq_{\text{rpo}} t$, for some $1 \leq i \leq m$ or

3. $f = g$, $\mathrm{Stat}(f) = \mathrm{lex}, m = n$, and either one of the following properties holds:

   (a) $s_i \succ_{\mathrm{rpo}} t$ or $s_i \simeq_{\mathrm{rpo}} t$, for some $1 \leq i \leq m$

   (b) $\langle s_1, \ldots, s_m \rangle \succ_{\mathrm{rpo}}^{\mathrm{lex}} \langle t_1, \ldots, t_n \rangle$ and $s \succ_{\mathrm{rpo}} t_j$, for all $1 \leq j \leq n$

   or

4. $f = g$, $\mathrm{Stat}(f) = \mathrm{mul}$ and $\{s_1, \ldots, s_m\} \succ_{\mathrm{rpo}}^{\mathrm{mul}} \{t_1, \ldots, t_n\}$

**Proposition 2.3.2**
RPO is a quasi-ordering, because it is total only if $\simeq$ is interpreted as equality up to permutations of arguments. RPO is a rewrite quasi-ordering and a simplification quasi-ordering. RPO is a reduction quasi-ordering if $\mathcal{F}$ is finite. For precedences without multiset function symbols RPO is equivalent to LPO and hence Proposition 2.3.1 holds.

## 2.4 Ordering Constraints

**Definition 2.4.1 (Ordering Constraint)**
An *ordering constraint* is a quantifier-free first-order formula built over terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ and the binary predicate symbols $\{\succ, \simeq\}$ (or $\{\succ, \succeq, \simeq\}$).

**Definition 2.4.2 (Solution)**
A *solution* of a constraint $\mathcal{C}$ is a substitution $\sigma$ with range $\mathcal{T}(\mathcal{F})$ and the domain the set of free variables of $\mathcal{C}$ such that $\mathcal{C}\sigma$ evaluates to true when interpreting $\simeq$ as equality on terms (equivalence classes for RPO ordering constraints) and $\succ$ as ordering on terms. If there is a solution $\sigma$ for a given constraint $\mathcal{C}$ then $\mathcal{C}$ is said to be *satisfiable*.

# 3

# RPO Constraints

## 3.1 The Problem

The goal of this thesis is to develop and implement an algorithm to check the satisfiability of RPO constraints and find *one* solution if the constraint is satisfiable. Extending known algorithms, we allow arbitrary precedences and signatures. Moreover, arbitrary–arity multiset–status function symbols are handled.

For LPO constraints, H. Comon proved in [Com90b] that satisfiability is decidable, but his algorithm was designed to fit a nice decidability and is not very practicable or efficient. For this reason A. Rubio and R. Nieuwenhuis introduced a simpler and more efficient algorithm in [RN91]. This algorithm relies on a restriction on precedences: only precedences with $0 < f < \dots$ are allowed, where 0 is the smallest constant and $f$ the smallest non-constant function symbol. This ensures that the *successor function* (see Def. 3.2.4) is total on terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$. We will see later how this affects our algorithm and how this restriction can be dropped.

J.P. Jouannaud and M. Okada showed in [JO91] that the satisfiability of RPO constraints is decidable. Again, the algorithm given there to construct the proof is neither practicable nor efficient. In [Wei94] Ch. Weidenbach introduced an algorithm that applies the methods of [RN91] to RPO constraints and drops the restrictions on the precedence at the same time. Furthermore, this algorithm considers RPO constraints as formulas over the predicate symbols $\simeq, \succ$ and also $\succeq$. This approach (also discussed in [Rub94a] for LPO) is more efficient, as it reduces the number of generated solved problems for a given constraint, and it makes the algorithm more usable for practical purposes. The algorithm presented here is a modified version of [Wei94], where I adopted the rewrite rules and the definition of the successor function to handle arbitrary–arity multiset–status function symbols.

## 3.2 The Algorithm

The algorithm works in two steps. The first step uses a set of rewrite rules to compute the *disjunctive normal form (DNF)* of the input constraint and to then rewrite the DNF into a disjunction of *solved problems*. In the second step, the solved problems are tested for satisfiability — one after another, until one is found satisfiable or all of them have been tested without success.

Let $\Sigma = (\mathcal{X}, \mathcal{F}, \mathcal{P})$ be a signature with $\mathcal{X}$ a set of variables, $\mathcal{P} = \{\succ, \succeq, \simeq\}$ and $\mathcal{F}$ the set of function symbols. Every function symbol in $\mathcal{F}$ has a *status*, which can be mul or lex, denoted by $\mathrm{Stat}(f) = \mathrm{mul}$ or $\mathrm{Stat}(f) = \mathrm{lex}$ (see Def. 2.3.12). The total precedence on $\mathcal{F}$ is denoted by '$>$'. The smallest constant with respect to '$>$' is denoted by $0 \in \mathcal{F}$, the smallest non–constant function symbol by $f \in \mathcal{F}$. $\mathcal{C} = \{c_i \in \mathcal{F}_0 | c_i < f, c_i \neq 0\}$ is the set of all constants smaller than $f$, without 0. If $0 < f$ and $\mathcal{C} = \varnothing$ the precedence is called *simple*. The input for the algorithm is an RPO constraint, defined below.

**Definition 3.2.1 (RPO Constraint)**
An RPO constraint is a quantifier–free first–order formula built over the binary predicate symbols '$\succ$', '$\succeq$' and '$\simeq$' relating terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$. The RPO constraint is interpreted over the initial ground term algebra $\mathcal{T}(\mathcal{F})$. To decide the satisfiability, '$\succ$' is interpreted as the recursive path ordering '$\succ_{\mathrm{rpo}}$' and '$\simeq$' as its associated congruence '$\simeq_{\mathrm{rpo}}$'. ('$\succeq$' is interpreted as '$\succeq_{\mathrm{rpo}}$', which is defined in the usual way: $s \succeq_{\mathrm{rpo}} t \Leftrightarrow s \succ_{\mathrm{rpo}} t \vee s \simeq_{\mathrm{rpo}} t$)

We will use '$\succ$', '$\succeq$' and '$\simeq$' when dealing with syntax and '$\succ_{\mathrm{rpo}}$', '$\succeq_{\mathrm{rpo}}$' and '$\simeq_{\mathrm{rpo}}$' when dealing with semantics.
**Note:**
From now on, we use # as an abbreviation for '$\succ$ or $\succeq$', i.e. $\# \in \{\succ, \succeq\}$.

Given some constraint, we can now compute its disjunctive normal form by exhaustive application of the rules in Table 3.1.

The resulting constraint has the form $P_1 \vee \cdots \vee P_n$ where each $P_i$ in turn has the form $L_1 \wedge \cdots \wedge L_k$. Each *literal* $L_j$ is either some $t \simeq s$ or some $t\#s$. The $P_i$'s are called *problems*. $P_{\simeq}$ denotes the restriction of a problem $P$ to atoms $s \simeq t$.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1. | $\neg\top \to \bot$ | | 9. | $s \simeq s$ | $\to$ | $\top$ |
| 2. | $\neg\bot \to \top$ | | 10. | $s \succ s$ | $\to$ | $\bot$ |
| 3. | $a \wedge a \to a$ | | 11. | $\neg(s \succ t)$ | $\to$ | $t \succeq s$ |
| 4. | $a \vee a \to a$ | | 12. | $\neg(s \succeq t)$ | $\to$ | $t \succ s$ |
| 5. | $a \wedge \top \to a$ | | 13. | $\neg(s \simeq t)$ | $\to$ | $s \succ t \vee t \succ s$ |
| 6. | $a \vee \top \to \top$ | | 14. | $\neg(a_1 \vee a_2)$ | $\to$ | $\neg a_1 \wedge \neg a_2$ |
| 7. | $a \wedge \bot \to \bot$ | | 15. | $\neg(a_1 \wedge a_2)$ | $\to$ | $\neg a_1 \vee \neg a_2$ |
| 8. | $a \vee \bot \to a$ | | 16. | $(a_1 \vee a_2) \wedge a_3$ | $\to$ | $(a_1 \wedge a_3) \vee (a_2 \wedge a_3)$ |

Table 3.1: Normalization of Literals and Propositional Normalization

**Definition 3.2.2**
A formula $t_1 \simeq s_1 \wedge \cdots \wedge t_k \simeq s_k$ is called *solved* if

$$\forall i : t_i \text{ variable}, \forall i, j, i \neq j : t_i \neq t_j \text{ and } \forall i, j \geq i : t_i \notin \mathit{Vars}(s_j)$$

We will next give a set of rules $\mathcal{R}$ for the transformation of inequational formulas. The corresponding reduction relation is denoted $\to_{\mathcal{R}}$. $\mathcal{R}$ is called *correct* if $\phi \to_{\mathcal{R}} \phi'$ implies that $\phi$ and $\phi'$ have the same solution. It is called *complete* (wrt. a given set of solved problems) if any normal form for $\to^*_{\mathcal{R}}$ is a solved problem. And finally, $\mathcal{R}$ is said to be *terminating* if there is no infinite sequence $F_1 \to_{\mathcal{R}} F_2 \to_{\mathcal{R}} \ldots$ of inequational formulas.

The set of rules $\mathcal{R}$ is shown in Tables 3.2, 3.3 and 3.4. As the rules apply to disjunctive normal forms of problems, we assume that the rules in Table 3.1 have been applied exhaustively before each application of a rule in $\mathcal{R}$.

The rule set $\mathcal{R}$ consists of two parts: Table 3.2 shows the rules for syntactic unification to handle the equational part of a problem (for an overview over syntactic unification and its extensions to equational theory see [Sie89]). The second part consists of the RPO substitution rules in Table 3.3 and the RPO decomposition rules in Table 3.4 to handle the inequational part of a problem.

The exhaustive application of the rules in $\mathcal{R}$ transforms the constraint into a disjunction of *solved problems*.

| |
|---|
| Tautology |
| $\quad t \simeq t \wedge P \rightarrow P$ |
| Decomposition-lex |
| $\quad f(t_1, \ldots, t_n) \simeq f(s_1, \ldots, s_n) \wedge P \rightarrow t_1 \simeq s_1 \wedge \ldots \wedge t_n \simeq s_n \wedge P$ |
| $\quad$ if $\mathrm{Stat}(f) = \mathrm{lex}$ |
| Decomposition-mul |
| $\quad f(t_1, \ldots, t_n) \simeq f(s_1, \ldots, s_n) \wedge P \rightarrow \bigvee_{\pi \in S^n} \left[ \left( \bigwedge_{1 \le j \le n} t_j \simeq s_{\pi(j)} \right) \wedge P \right]$ |
| $\quad$ if $\mathrm{Stat}(f) = \mathrm{mul}$ |
| $\quad f(t_1, \ldots, t_m) \simeq f(s_1, \ldots, s_n) \wedge P \rightarrow \perp$ |
| $\quad$ if $\mathrm{Stat}(f) = \mathrm{mul}$, $m \neq n$ |
| Substitution |
| $\quad x \simeq y \wedge P \rightarrow x \simeq y \wedge P\{x/y\}$ |
| $\quad$ if $x \in \mathit{Vars}(P)$ and $y \in \mathit{Vars}(P)$ |
| Clash |
| $\quad f(t_1, \ldots, t_m) \simeq g(s_1, \ldots, s_m) \wedge P \rightarrow \perp$ |
| $\quad$ if $f \neq g$ |
| Cycle |
| $\quad x_1 \simeq t_1[x_2]_{p_1} \wedge \ldots \wedge x_n \simeq t_n[x_1]_{p_n} \wedge P \rightarrow \perp$ |
| $\quad$ if there exists some $n$ and $i$, $1 \le i \le n$, with $p_i \neq \lambda$ |
| Merge |
| $\quad x \simeq t \wedge x \simeq s \wedge P \rightarrow x \simeq t \wedge t \simeq s \wedge P$ |
| $\quad$ if $\mathrm{size}(t) \le \mathrm{size}(s)$ |

Table 3.2: The Rules of Standard Unification

**Definition 3.2.3 (Solved Problem)**
A *solved problem* $P$ is either $\top$, $\perp$ or a formula

$$x_1 \# t_1 \wedge \cdots \wedge x_n \# t_n \wedge t'_1 \# x'_1 \wedge \cdots \wedge t'_m \# x'_m \wedge y_1 \simeq s_1 \wedge \cdots \wedge y_k \simeq s_k$$

with $P_{\simeq}$ solved, no $t'_j$ is a variable, $t'_j \neq 0$, $x_i \notin \mathit{Vars}(t_i)$ , $x'_j \notin \mathit{Vars}(t'_j)$, $y_l \neq x_i$, $y_l \neq t_i$, $y_l \neq x'_j$, $1 \le i \le n$, $1 \le j \le m$, $1 \le l \le k$.

The terms $t_i$ are called *right terms* in $P$, the atoms $x_i \# t_i$ are called *right atoms*. Terms $t'_j$ and atoms $t'_j \# x'_j$ are correspondingly called *left terms* and *left atoms* in $P$.

**Example**
To get a feeling for the complexity of the problem, we will examine a small

example. Given the function symbols $f$, $g$ and $h$ with arities 3, 2 and 1 respectively, $\mathrm{Stat}(f) = \mathrm{mul}$, $\mathrm{Stat}(g) = \mathrm{Stat}(h) = \mathrm{lex}$ and the constants 0, $a$, $b$ and $c$, a precedence $h > g > f > c > b > a > 0$ and an input constraint

$$f(g(a,x),b,y) \succeq f(h(x),a,0) \wedge g(0,x) \succ g(a,y)$$

the rule *Decomposition-mul-left* can be applied on the first inequation:

$$\begin{aligned}
&f(g(a,x),b,y) \succeq f(h(x),a,0) \wedge g(0,x) \succ g(a,x) \rightarrow \\
&\quad [g(a,x) \succeq b \wedge b \succeq y \wedge h(x) \succeq a \wedge a \succeq 0 \wedge \\
&\quad (\ \ (g(a,x) \succ h(x) \wedge g(0,x) \succ g(a,x)) \\
&\quad \vee (g(a,x) \simeq h(x) \wedge b \succeq a \wedge g(0,x) \succ g(a,x)) \\
&\quad \vee (g(a,x) \simeq h(x) \wedge b \simeq a \wedge y \succeq 0 \wedge g(0,x) \succ g(a,x)))] \vee \\
&\quad [g(a,x) \succeq b \wedge b \succeq y \wedge h(x) \succeq 0 \wedge 0 \succeq a \wedge \\
&\quad (\ \ (g(a,x) \succ h(x) \wedge g(0,x) \succ g(a,x)) \\
&\quad \vee (g(a,x) \simeq h(x) \wedge b \succeq 0 \wedge g(0,x) \succ g(a,x)) \\
&\quad \vee (g(a,x) \simeq h(x) \wedge b \simeq 0 \wedge y \succeq a \wedge g(0,x) \succ g(a,x)))] \vee \\
&\quad \ldots \text{32 more conjunctions} \ldots \\
&\quad [y \succeq b \wedge b \succeq g(a,x) \wedge 0 \succeq h(x) \wedge h(x) \succeq a \wedge \\
&\quad (\ \ (y \succ 0 \wedge g(0,x) \succ g(a,x)) \\
&\quad \vee (y \simeq 0 \wedge b \succeq h(x) \wedge g(0,x) \succ g(a,x)) \\
&\quad \vee (y \simeq 0 \wedge b \simeq h(x) \wedge g(a,x) \succeq a \wedge g(0,x) \succ g(a,x)))] \vee \\
&\quad [y \succeq b \wedge b \succeq g(a,x) \wedge 0 \succeq a \wedge a \succeq h(x) \wedge \\
&\quad (\ \ (y \succ 0 \wedge g(0,x) \succ g(a,x)) \\
&\quad \vee (y \simeq 0 \wedge b \succeq a \wedge g(0,x) \succ g(a,x)) \\
&\quad \vee (y \simeq 0 \wedge b \simeq a \wedge g(a,x) \succeq h(x) \wedge g(0,x) \succ g(a,x)))]
\end{aligned}$$

Another choice is to apply the rule *Decomposition-lex* to the second inequation of the input constraint:

$$\begin{aligned}
&f(g(a,x),b,y) \succeq f(h(x),a,0) \wedge g(0,x) \succ g(a,x) \rightarrow \\
&\ \ (0 \succ a \wedge g(0,x) \succ a \wedge g(0,x) \succ x \wedge f(g(a,x),b,y) \succeq f(h(x),a,0)) \\
&\vee (0 \simeq a \wedge x \succ x \wedge g(0,x) \succ a \wedge g(0,x) \succ x \\
&\qquad\qquad \wedge f(g(a,x),b,y) \succeq f(h(x),a,0)) \\
&\vee (0 \succeq g(a,x) \wedge f(g(a,x),b,y) \succeq f(h(x),a,0)) \\
&\vee (x \succeq g(a,x) \wedge f(g(a,x),b,y) \succeq f(h(x),a,0))
\end{aligned}$$

Here, the first conjunction reduces to $\perp$ by application of *Decomposition-left* on $0 \succ a$, the second conjunction reduces to $\perp$ by application of *Clash* on $0 \simeq a$, the third one reduces to $\perp$ by application of *Decomposition-left*

| |
|---|
| Substitution |
| $x \# s \wedge x \simeq t \wedge P \to t \# s \wedge x \simeq t \wedge P$ |
| $s \# x \wedge x \simeq t \wedge P \to s \# t \wedge x \simeq t \wedge P$ |
| if $(x \simeq t \wedge P_{\simeq})$ is solved |
| Tautology |
| $t[s]_p \# s \wedge P \to P$ |
| if $p \neq \lambda$ or $\# = \succeq$ |
| Cycle |
| $t_1 \# s_1[t_2]_{p_1} \wedge \ldots \wedge t_n \# s_n[t_1]_{p_n} \wedge P \to \bot$ |
| if some $\# = \succ$ or some $p_i \neq \lambda$ |
| Simplification |
| $0 \succ x \wedge P \to \bot$ |
| $0 \succeq x \wedge P \to P\{x/0\}$ |
| $s \succ t \wedge s \succeq t \wedge P \to s \succ t \wedge P$ |
| $s \simeq t \wedge s \succeq t \wedge P \to s \simeq t \wedge P$ |

Table 3.3: The Rules of RPO Substitution and Simplification

on $0 \succeq g(a, x)$ and the last one reduces to $\bot$ by application of *Cycle* on $x \succeq g(a, x)$. Hence the constraint is unsatisfiable.

This small example illustrates two aspects of the rewrite system: some of the rewrite rules produce *huge* amounts of new equations and inequations, and the order in which the rules are applied is very important for the efficiency of the constraint solver.

**Lemma 3.2.1**
The rules in Tables 3.2, 3.3 and 3.4 together are correct, complete and terminating.
**Proof:**
(For the proof technique see [Com90a]) Correctness is a direct consequence of the definition of $\succ_{\text{rpo}}$: Decomposition-mul in Table 3.2 corresponds to the definition of equivalence on multisets (see Def. 2.3.7), Decomposition-right in Table 3.4 corresponds to Def. 2.3.12(1), Decomposition-left in Table 3.4 corresponds to Def. 2.3.12(2), Decomposition-lex in Table 3.4 to Def. 2.3.12(3) and Decomposition-mul(-left,-right) in Table 3.4 to Def. 2.3.12(4). The correctness of the other rules is obvious.

Completeness is easy to check, if the rewrite system terminates. So let us prove termination now — we use the following interpretation functions:

- $\Phi_1(s_1 \simeq t_1 \wedge \ldots \wedge s_n \simeq t_n \wedge u_1 \# v_1 \wedge \ldots \wedge u_m \# v_m)$ is the multiset of

Decomposition-right

$$f(t_1,\ldots,t_m)\#g(s_1,\ldots,t_n) \wedge P \rightarrow \left(\bigwedge_{1\leq j\leq n} f(t_1,\ldots,t_m) \succ s_j\right) \wedge P$$

if $f > g$

Decomposition-left

$$f(t_1,\ldots,t_m)\#g(s_1,\ldots,s_n) \wedge P \rightarrow \left(\bigvee_{1\leq i\leq m} t_i \succeq g(s_1,\ldots,s_n) \wedge P\right)$$

if $f < g$

Decomposition-lex

$$f(t_1,\ldots,t_n)\#f(s_1,\ldots,s_n) \wedge P \rightarrow$$
$$\bigvee_{1\leq i<n}\left[\left(\bigwedge_{1\leq j<i} t_j \simeq s_j\right) \wedge t_i \succ s_i \wedge \left(\bigwedge_{i<k\leq n} f(t_1,\ldots,t_n) \succ s_k\right) \wedge P\right] \vee$$
$$\left[\left(\bigwedge_{1\leq j<n} t_j \simeq s_j\right) \wedge t_n \# s_n \wedge P\right] \vee \left(\bigvee_{1\leq i\leq n} t_i \succeq f(s_1,\ldots,s_n) \wedge P\right)$$

if $\mathrm{Stat}(f) = \mathrm{lex}$

Decomposition-mul-left

$$f(t_1,\ldots,t_m)\#f(s_1,\ldots,s_n) \wedge P \rightarrow$$
$$\bigvee_{\pi\in S^m}\bigvee_{\kappa\in S^n}\Bigg[t_{\pi(1)} \succeq \ldots \succeq t_{\pi(m)} \wedge s_{\kappa(1)} \succeq \ldots \succeq s_{\kappa(n)} \wedge$$
$$\bigvee_{1\leq i<n}\left[\left(\bigwedge_{1\leq j<i} t_{\pi(j)} \simeq s_{\kappa(j)}\right) \wedge t_{\pi(i)} \succ s_{\kappa(i)} \wedge P\right] \vee$$
$$\left[\left(\bigwedge_{1\leq j<n} t_{\pi(j)} \simeq s_{\kappa(j)}\right) \wedge t_{\pi(n)} \# s_{\kappa(n)} \wedge P\right]\Bigg]$$

if $\mathrm{Stat}(f) = \mathrm{mul}$, $m \geq n$

Decomposition-mul-right

$$f(t_1,\ldots,t_m)\#f(s_1,\ldots,s_n) \wedge P \rightarrow$$
$$\bigvee_{\pi\in S^m}\bigvee_{\kappa\in S^n}\Bigg[t_{\pi(1)} \succeq \ldots \succeq t_{\pi(m)} \wedge s_{\kappa(1)} \succeq \ldots \succeq s_{\kappa(n)} \wedge$$
$$\bigvee_{1\leq i<m}\left[\left(\bigwedge_{1\leq j<i} t_{\pi(j)} \simeq s_{\kappa(j)}\right) \wedge t_{\pi(i)} \succ s_{\kappa(i)} \wedge P\right] \vee$$
$$\left[\left(\bigwedge_{1\leq j<m} t_{\pi(j)} \simeq s_{\kappa(j)}\right) \wedge t_{\pi(m)} \succ s_{\kappa(m)} \wedge P\right]\Bigg]$$

if $\mathrm{Stat}(f) = \mathrm{mul}$, $m < n$

Table 3.4: The Rules of RPO

multisets of natural numbers:

$$\{\{|s_1|, |t_1|\}, \ldots, \{|s_n|, |t_n|\}, \{|u_1|, |v_1|\}, \ldots, \{|u_m|, |v_m|\}\}$$

where $|s|$ is the number of function symbols and variables occurring in $s$ (size of $s$). Such multisets are ordered by the usual multiset extension of $>$ on $\mathbb{N}$ (See Def. 2.3.8).

- $\Phi_2(s_1 \simeq t_1 \wedge \ldots \wedge s_n \simeq t_n \wedge u_1 \# v_1 \wedge \ldots \wedge u_m \# v_m)$ is the number of *unsolved variables* in the system. A variable $x$ is solved in such a system if $x$ is a member of an equation $x = t$ and $x$ occurs only once in the system. (Note: For proof purposes this definition of *solved* is different from the Definitions 3.2.3 and 3.2.2.)

- $\Phi(\bigvee_{1 \le j \le n} P_j)$, where $P_j$ is a conjunction of equations and inequations, is the multiset of pairs $\langle \Phi_2(P_j), \Phi_1(P_j) \rangle$. Such interpretations are ordered using the multiset extension of the lexicographic extension of $>$ to pairs.

We will prove that $\Phi$ is strictly decreasing by application of any rule to an RPO constraint (in DNF) or (in some exceptions described below) by application of a rule *and* any possible sequence of following applications of rules to the constraint.

Assume that the constraint has the form $P \vee \bigvee_{1 \le j \le n} P_j$ and $P \rightarrow \bigvee_{1 \le i \le m} P'_i$. Now we have to prove that, for every $1 \le i \le m$, either $\Phi_2(P'_i) < \Phi_2(P)$ or $\Phi_2(P'_i) = \Phi_2(P)$ and $\Phi_1(P'_i) < \Phi_1(P)$. Note that $\Phi_2(P'_i) \le \Phi_2(P)$ for every rule, because there's no rule which turns a solved variable into an unsolved variable. Let us check now the application of the rules using *Decomposition-lex* in Table 3.2 as an example. In the following, $P$ denotes the conjunctive subproblem to which the rule is applied:
*(Decomposition-lex)*:

$$P \equiv P' \wedge f(\vec{t}) \simeq f(\vec{u}) \rightarrow P\text{''} \equiv P' \wedge \bigwedge_{1 \le i \le m} t_i \simeq u_i$$

$$\Phi_1(P) = \{a_1, \ldots, a_k, \{1 + b_1 + \cdots + b_m, 1 + c_1 + \cdots + c_m\}\}$$

with $a_i$ size of the rest problem terms, $b_i = |t_i|$ and $c_i = |u_i|$.

$$\Phi_1(P\text{''}) = \{a_1, \ldots, a_k, \{b_1, c_1\}, \ldots, \{b_m, c_m\}\}$$

$$\Rightarrow \Phi_1(P\text{''}) < \Phi_1(P). \text{ (By definition of } >_{\text{mul}} \text{ on } \mathbb{N}\text{).}$$

The proof is similar for all the other *Decomposition* rules in Tables 3.2 and 3.3: in all these cases the original equation or inequation is removed and

(possibly many) new equations and/or inequations are added with subterms of the original equation or inequation on one or both sides. Thus $\Phi_1$ decreases. The *Tautology* rules in both tables and the last three rules in *Simplification* in Table 3.3 remove one equation or inequation and thus reduce $\Phi_1$. The two *Cycle* rules, *Clash* and the first rule in *Simplification* in Table 3.3 reduce the whole problem to $\perp$, and so $\Phi_1$ decreases as $\Phi_1(\perp) = \{\{0\}\}$.
This leaves us with the *Substitution* rules and the *Merge* rules. Let us first look at *Merge*:
*(Merge)*:

$$P \equiv P' \wedge x \simeq t \wedge x \simeq s \rightarrow P'' \equiv P' \wedge x \simeq t \wedge t \simeq s \quad \text{if size}(t) \leq \text{size}(s)$$

This rule increases $\Phi_1$:

$$\begin{aligned}
&\Phi_1(P) = \{a_1, \ldots, a_k, \{1, |t|\}, \{1, |s|\}\}, \\
&\Phi_1(P'') = \{a_1, \ldots, a_k, \{1, |t|\}, \{|t|, |s|\}\}
\end{aligned}$$

But apart from rules that reduce $\Phi_1$ drastically, the only rules that can be applied to the new equation $t \simeq s$ are the equational *Decomposition* rules. So the worst case for applying *Merge* and then another rule to $t \simeq s$ is that $\{|s|, |t|\}$ is replaced by (possibly many) $\{|s_i|, |t_j|\}$. Due to the condition $\text{size}(t) \leq \text{size}(s)$ all those $\{|s_i|, |t_j|\}$ multisets are smaller than $\{1, |s|\}$ in $\Phi_1(P)$, and thus $\Phi_1$ decreases.

Finally the *Substitution* rules: *Substitution* in Table 3.2 removes all occurrences (but one) of $x$ in the problem and thereby decreases $\Phi_2$. The two *Substitution* rules in Table 3.3 are similar to *Merge*, with the difference that the RPO *Decomposition* rules have to be applied repeatedly.

This proves the strict decreasingness of $\Phi$ by application of any rule (or by application of a rule and then any possible sequence of following applications of rules). Since the inequational problems are interpreted by $\Phi$ in a well-founded domain, this proves termination.

Finally we'll prove the completeness of the rules. The normalization rules in 3.2 keep the constraint in the following form:

$$\mathcal{C} \equiv \bigvee_{1 \leq i \leq l} P_i = \bigvee_{1 \leq i \leq l} \left( \left( \bigwedge_{1 \leq j \leq m} s_{ij} \simeq t_{ij} \right) \wedge \left( \bigwedge_{1 \leq k \leq n} u_{ik} \# v_{ik} \right) \right)$$

Now assume that the repeated application of the rules terminates, producing $\mathcal{C} \equiv \bigvee_{1 \leq i \leq l} P_i$. If any of the $P_i$ is not a solved problem as in Def. 3.2.3, one of the following cases applies:

1. $u_{ik} = f(\vec{u}) \# g(\vec{v}) = v_{ik}$ for some $k$
   $\Rightarrow$ One of the *Decomposition* rules in Table 3.4 or *Tautology*, *Cycle* or *Simplification* in Table 3.3 can be applied.

2. $P_{i\simeq}$ not solved

(a) some $y_l$ not a variable
$\Rightarrow$ *Tautology*, *Clash* or some *Decomposition* rule in Table 3.3 applies.

(b) some $y_l = y_{l'}$ for $l \neq l'$
$\Rightarrow$ *Merge* in Table 3.2 can be applied.

(c) some $y_l \in Vars(s_{l'})$ for $l \neq l'$:
$\Rightarrow y_l \simeq s_l \wedge y_{l'} \simeq s_{l'} \wedge P_i$ with $y_l \in Vars(s_{l'})$
If the equations in $P_{i\simeq}$ cannot be rearranged s.t. this condition is not violated, then *Cycle* in Table 3.3 can be applied.

3. Some $t'_j$ is a variable
$\Rightarrow$ No violation, the inequation then is part of the $x_i \# t_i$.

4. Some $t'_j = 0$
$\Rightarrow$ The first or second rule in *Simplification* in Table 3.4 can be applied.

5. Some $x_i \in Vars(t_i)$
$\Rightarrow$ Rule ($s \succ s \rightarrow \bot$) (Table 3.2), *Tautology* (Table 3.4) or *Cycle* (Table 3.4) applies.

6. Some $x'_j \in Vars(t'_j)$
$\Rightarrow$ *Tautology* in Table 3.4 applies.

7. Some $y_l = t_i$: $x_i \# y_l \wedge y_l \simeq s_l \wedge P_i$
$\Rightarrow$ *Substitution* in Table 3.4 applies.

8. Some $y_l = x'_j$: $t'_j \# y_l \wedge y_l \simeq s_l \wedge P_i$
$\Rightarrow$ *Substitution* in Table 3.4 applies.

As the system terminates, this proves completeness. □

Now after applying the normalization, unification and RPO rules the constraint is a disjunction of solved problems, hence an RPO constraint is satisfiable if and only if one of its solved problems is satisfiable. Unfortunately, it is not easy to check satisfiability of solved problems.

**Lemma 3.2.2**
Let $P = t'_1 \# x'_1 \wedge \cdots \wedge t'_m \# x'_m$ be a solved problem, no $t'_j$ is a variable, then $P$ is solved by a substitution $\sigma$ with $x\sigma = 0$ for all $x \in Vars(P)$.
**Proof:** Obvious. □

Due to the definition of solved problems (see Def. 3.2.3) and Lemma 3.2.2 it's easy to see that only right atoms play a role for the satisfiability of a solved problem. The idea now is to eliminate all right atoms and then (if the solved problem is satisfiable) compute a solution by setting all $x'_j$'s to 0 and compute the substitution for the $y_l$'s from right to left.

The method in [RN91] (for LPO constraints) to eliminate right atoms works as follows: replace $x_i \succ t_i$ for the maximal right term $t_i$ with $x_i \simeq \mathrm{succ}(t_i)$. As we don't know the maximal right term, all right terms have to be tried. This approach works only if the successor function is total, which is not the case for RPO.

**Definition 3.2.4** (succ)
Let $t = g(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $s = g(s_1, \ldots, s_n) \in \mathcal{T}(\mathcal{F})$. Recall: 0 is the smallest constant in $\mathcal{F}$, $f$ is the smallest non-constant function symbol in $\mathcal{F}$, $\mathcal{C} = \{c_i \in \mathcal{F}_0 \mid c_i < f,\ c_i \neq 0\}$ and $|\mathcal{C}| = m$. Now the successor function succ is defined as follows:

$$\mathrm{succ}(0) = \begin{cases} c_1 & \text{if } \mathcal{C} \neq \varnothing \\ f(\vec{0}) & \text{if } \mathcal{C} = \varnothing,\ \mathrm{arity}(f) \text{ fixed} \\ f(0) & \text{if } \mathcal{C} = \varnothing,\ \mathrm{arity}(f) \text{ arbitrary} \end{cases}$$

$$\mathrm{succ}(c_i) = \begin{cases} c_{i+1} & \text{if } i < m \\ f(\vec{0}) & \text{if } i = m,\ \mathrm{arity}(f) \text{ fixed} \\ f(0) & \text{if } i = m,\ \mathrm{arity}(f) \text{ arbitrary} \end{cases}$$

$$\mathrm{succ}(s) = \begin{cases} f(\vec{0}, \mathrm{succ}(s_n)) & \text{if } \mathrm{Stat}(f) = \mathrm{lex},\ g = f, \\ & \quad s_1 = \ldots = s_{n-1} = 0 \\ f(\vec{0}, s) & \text{if } \mathrm{Stat}(f) = \mathrm{lex}, \text{ otherwise} \\ f(\vec{s_k}, \mathrm{succ}(s_{k+1}), \vec{0}) & \text{if } \mathrm{Stat}(f) = \mathrm{mul},\ g = f, \\ & \quad \mathrm{arity}(f) \text{ fixed} \\ f(\vec{s_k}, \mathrm{succ}(s_{k+1})) & \text{if } \mathrm{Stat}(f) = \mathrm{mul},\ g = f, \\ & \quad \mathrm{arity}(f) \text{ arbitrary} \\ f(s, \vec{0}) & \text{if } \mathrm{Stat}(f) = \mathrm{mul},\ g \neq f, \\ & \quad \mathrm{arity}(f) \text{ fixed} \\ f(s) & \text{if } \mathrm{Stat}(f) = \mathrm{mul},\ g \neq f, \\ & \quad \mathrm{arity}(f) \text{ arbitrary} \end{cases}$$

$$\mathrm{succ}(t) = \begin{cases} f(\vec{0}, t) & \text{if } \mathrm{Stat}(f) = \mathrm{lex},\ > \text{simple} \\ f(t, \vec{0}) & \text{if } \mathrm{Stat}(f) = \mathrm{mul},\ g \neq f, \\ & \quad \mathrm{arity}(f) \text{ fixed} \\ f(t) & \text{if } \mathrm{Stat}(f) = \mathrm{mul},\ g \neq f, \\ & \quad \mathrm{arity}(f) \text{ arbitrary} \\ \text{undefined} & \text{otherwise} \end{cases}$$

where we assume for the third and fourth case of $\mathrm{succ}(s)$ that the $s_i$ are sorted in descending order wrt. $\succ_{\mathrm{rpo}}$ and $s_{k+1}$ is the leftmost subterm of $g(s_1, \ldots, s_n)$ with $s_i \simeq s_{i+1}$ for all $k < i < n$.

**Lemma 3.2.3**
Let $t$ be a term with $t \in \mathrm{dom}(\mathrm{succ})$. There is no term $t'$ with $\mathrm{succ}(t) \succ_{\mathrm{rpo}} t' \succ_{\mathrm{rpo}} t$.
**Proof:** We proceed by induction on the size of $t$ wrt. $\succ^{\mathcal{F}}_{\mathrm{rpo}}$. The first three cases for $\mathrm{succ}(0)$ and the next three cases for $\mathrm{succ}(c_i)$ are obvious (due to $>^{\mathcal{F}}$) . Now let's look at the first case of $\mathrm{succ}(s)$: assume $\mathrm{succ}(s) = f(\vec{0}, \mathrm{succ}(s_n)) \succ^{\mathcal{F}}_{\mathrm{rpo}} t' \succ^{\mathcal{F}}_{\mathrm{rpo}} f(\vec{0}, s_n) = s$ for some $t'$. Then $f(\vec{0}, \mathrm{succ}(s_n)) \succ^{\mathcal{F}}_{\mathrm{rpo}} f(\vec{0}, t'') \succ^{\mathcal{F}}_{\mathrm{rpo}} f(\vec{0}, s_n)$ for some $t''$, and that implies $\mathrm{succ}(s_n) \succ^{\mathcal{F}}_{\mathrm{rpo}} t'' \succ^{\mathcal{F}}_{\mathrm{rpo}} s_n$, which is impossible by induction hypothesis. The proof for the other cases of $\mathrm{succ}(s)$ is analogous to this one. Finally let's look at the first case of $\mathrm{succ}(t)$: assume there is a $t'$ with $\mathrm{succ}(t) = f(\vec{0}, t) \succ^{\mathcal{F}}_{\mathrm{rpo}} t' \succ^{\mathcal{F}}_{\mathrm{rpo}} t$. From $f(\vec{0}, g(t_1, \ldots, t_n)) \succ^{\mathcal{F}}_{\mathrm{rpo}} t'$ follows either $g(t_1, \ldots, t_n) \succeq^{\mathcal{F}}_{\mathrm{rpo}} t'$ (because $f$ is the smallest function symbol), which is impossible as $g(t_1, \ldots, t_n) \succeq^{\mathcal{F}}_{\mathrm{rpo}} t' \succ^{\mathcal{F}}_{\mathrm{rpo}} g(t_1, \ldots, t_n)$ can be derived, or $t' = f(\vec{0}, t'') \succ^{\mathcal{F}}_{\mathrm{rpo}} g(t_1, \ldots, t_n)$ for some $t''$. Then $t'' \succeq^{\mathcal{F}}_{\mathrm{rpo}} g(t_1, \ldots, t_n)$ which is impossible as $f(\vec{0}, g(t_1, \ldots, t_n)) \succ^{\mathcal{F}}_{\mathrm{rpo}} f(\vec{0}, t'')$. The proofs for the other cases of $\mathrm{succ}(t)$ are similar again. □

For the following three lemmata we assume $P = x_1 \# t_1 \wedge \cdots \wedge x_n \# t_n \wedge t'_1 \# x'_1 \wedge \cdots \wedge t'_m \# x'_m$ is a solved problem. The equality part $y_1 \simeq s_1 \wedge \cdots \wedge y_k \simeq s_k$ is not mentioned, because it plays no role for the satisfiability of the solved problem since $y_l \neq x_i$, $y_l \neq t_i$ and $y_l \neq x'_j$ and variables $x_i$ are replaced only by terms which keep $P_{\simeq}$ solved.

**Lemma 3.2.4** ([Wei94])
Let $P$ be a solved problem with maximal right term $t_i$ occuring in right atom $x_i \# t_i$ and $t_i \in \mathrm{dom}(\mathrm{succ})$. Now

$$\begin{aligned} &P \text{ satisfiable } \Leftrightarrow \\ &R = P \setminus \{t'_j[x_i]_p \# x'_j \mid p \neq \lambda, \text{ for all } j\} \\ &\qquad \cup \{t_i \succeq t_j \mid \text{for all } j\} \\ &\qquad \cup \{x_i \simeq \mathrm{succ}(t_i)\} \end{aligned}$$

satisfiable.

**Proof:**
"$\Rightarrow$" Let $\tau$ be a solution for $P$. We construct a ground substitution $\tau'$ by $x_i\tau' = \mathrm{succ}(t_i)\tau$ and $y\tau' = y\tau$ otherwise. Now $\tau'$ is a solution for $R$: obviously $x_i\tau' \simeq_{\mathrm{rpo}} \mathrm{succ}(t_i)\tau'$. Since $t_i$ is maximal and occurs in right atom $x_i \succ t_i$, $x_i \notin \mathit{Vars}(t_j)$, for all $j$ and thus $t_j\tau = t_j\tau'$ for all $j$. This proves that $\tau'$ satisfies all atoms $t_i \succeq t_j$ and all right atoms. Since $x_i\tau \succeq_{\mathrm{rpo}} x_i\tau'$ and there are no left terms including $x_i$ in $R$, $\tau'$ satisfies all left term literals.

"$\Leftarrow$" Assume some substitution $\tau$ satisfies $R$. Now we construct a substitution $\tau'$ satisfying both $R$ and $P$: $y\tau' = y\tau$ if $x_i\tau \succeq_{\text{rpo}} y\tau$ and $y\tau' = x_i\tau$ otherwise. First, we show that $\tau'$ satisfies $R$: by construction $x_i\tau' \simeq_{\text{rpo}} \text{succ}(t_i)\tau'$. Since $t_i$ is maximal wrt. $\tau$, $t_j\tau' = t_j\tau$ and $x_i\tau \succ_{\text{rpo}} t_i\tau \succeq_{\text{rpo}} t_j\tau$. This guarantees that $\tau'$ satisfies all right atoms and all atoms $t_i \succeq t_j$. For the left atoms, if $t'_j\tau \succ_{\text{rpo}} x_i\tau$ for some $j$, then $t'_j\tau' \succ_{\text{rpo}} x_i\tau'$, hence $\tau'$ satisfies $t'_j\#x'_j$. If $x_i\tau \succeq_{\text{rpo}} t'_j\tau$, then $t'_j\tau = t'_j\tau'$, $x'_j\tau = x'_j\tau'$, hence $\tau'$ satisfies $t'_j\#x'_j$. This completes the proof that $\tau'$ is a solution for $R$. In order to show that $\tau'$ satisfies $P$, it has to be shown that $\tau'$ satisfies the left atoms $t'_j[x_i]_p\#x'_j$ with $p \neq \lambda$. This is obvious since $t'_j[x_i]_p\tau' \succ_{\text{rpo}} x_i\tau' \succeq_{\text{rpo}} x'_j\tau'$. □

**Lemma 3.2.5** ([Wei94])
Let $P$ be a solved problem with maximal right term $t_i$ occurring in right atom $x_i \succ t_i$ and $t_i \notin \text{dom(succ)}$. Then

$$\begin{aligned} &P \text{ satisfiable } \Leftrightarrow \\ R = P &\setminus \{t'_j[x_i]_p\#x'_j \mid p \neq \lambda, \text{for all } j\} \\ &\setminus \{x_i\#t_j \mid \text{for all } j\} \\ &\cup \{t_i \succeq t_j \mid \text{for all } j\} \\ &\cup \{x_i \simeq t_i\} \end{aligned}$$

is satisfiable with ground substitution $\sigma$, and

$$\begin{aligned} Q = P &\setminus \{t'_j[x_i]_p\#x'_j \mid p \neq \lambda, \text{for all } j\} \\ &\cup \{t_i \succeq t_j \mid \text{for all } j\} \\ &\cup \{x_i \simeq \text{succ}(t_i\sigma)\} \end{aligned}$$

is satisfiable.

**Proof:**
"$\Rightarrow$" Let $\tau$ satisfy $P$. We construct a ground substitution $\sigma$ by $x_i\sigma = t_i\tau$ and $y\sigma = y\tau$ for all $y \neq x_i$. Now $\sigma$ satisfies $R$: obviously $x_i\sigma \simeq_{\text{rpo}} t_i\sigma$. Since $t_i$ is maximal and occurs in right atom $x_i \succ t_i$, $x_i \notin \mathit{Vars}(t_j)$, for all $j$ and thus $t_j\sigma = t_j\tau$ for all $j$. This proves that $\sigma$ satisfies all atoms $t_i \succeq t_j$ and all right atoms, because there is no right atom $x_i\#t_j$. Since $x_i\tau \succ_{\text{rpo}} x_i\sigma$ and there are no left terms including $x_i$ in $R$, $\sigma$ satisfies all left term literals. Now the substitution $\tau'$ with $x_i\tau' = \text{succ}(t_i\sigma)$ and $y\tau' = y\tau$ satisfies $Q$: since $x_i\tau' = \text{succ}(t_i\tau)$ and $y\tau' = y\tau$ this is already shown by the first part of Lemma 3.2.4.

"$\Leftarrow$" Assume that $\sigma$ satisfies $R$ and $\tau$ satisfies $Q$. We construct a substitution $\tau'$ by: $y\tau' = y\tau$ if $x_i\tau \succeq_{\text{rpo}} y\tau$ and $y\tau' = x_i\tau$ otherwise. Now we show that $\tau'$ satisfies both $Q$ and $P$ just as we did for the second part of Lemma 3.2.4. □

**Lemma 3.2.6** ([Wei94])
Let $P$ be a solved problem with maximal right term $t_i$ occurring in right atom $x_i \succeq t_i$ and there is no maximal right term $t_j$ in $P$ occurring in right atom $x_j \succ t_j$. Then

$$\begin{aligned} & P \text{ satisfiable } \Leftrightarrow \\ R = {} & P \setminus \{t'_j[x_i]_p \# x'_j \mid p \neq \lambda, \text{for all } j\} \\ & \cup \{t_i \succeq t_j \mid \text{for all } x_j \succ t_j \in P\} \\ & \cup \{t_i \succ t_j \mid \text{for all } x_j \succeq t_j \in P\} \\ & \cup \{x_i \simeq t_i\} \end{aligned}$$

is satisfiable.

**Proof:**
"$\Rightarrow$" Let $\tau$ satisfy $P$. We construct a ground substitution $\tau'$ by $x_i\tau' = t_i\tau$ and $y\tau' = y\tau$ otherwise. Now $\tau'$ satisfies $R$: obviously $x_i\tau' \simeq_{\text{rpo}} t_i\tau'$. Since $t_i$ is maximal and occurs in right atom $x_i \succeq t_i$, either $x_i \notin \mathit{Vars}(t_j)$ or $t_j = x_i$ and $x_j \succeq t_j \in P$ for all $j$. Since $x_i\tau \succeq x_i\tau'$, $\tau'$ satisfies all atoms $t_i \# t_j$ and all right atoms. For all left atoms $t'_j \# x'_j \in R$ we have $x_i \notin \mathit{Vars}(t'_j)$, hence $\tau'$ satisfies all left atoms in $R$.
"$\Leftarrow$" Assume that $\tau$ satisfies $R$. We construct a substitution $\tau'$ by: $y\tau' = y\tau$ if $x_i\tau \succeq_{\text{rpo}} y\tau$ and $y\tau' = x_i\tau$ otherwise. Now we show that $\tau'$ satisfies both $R$ and $P$. First, we show that $\tau'$ satisfies $R$: obviously $x_i\tau' \simeq_{\text{rpo}} t_i\tau'$ and $\tau'$ satisfies all atoms $t_i \# t_j$ because $t_j\tau' = t_j\tau$ for all $j$. For all right atoms $x_k \# t_k \in R$, either $t_i\tau \simeq_{\text{rpo}} t_k\tau$, whence by assumption $x_k \succeq t_k \in R$ and $x_k\tau' \simeq_{\text{rpo}} x_i\tau' \succeq_{\text{rpo}} t_k\tau'$ or $t_i\tau \succ_{\text{rpo}} t_k\tau$ and $x_k\tau' \succ_{\text{rpo}} t_k\tau'$. The rest of the proof follows the argumentation of the corresponding part in Lemma 3.2.4.
□

**Theorem 3.2.7**
The satisfiability of RPO constraints is decidable.
**Proof:**
Use the rewrite system in Tables 3.1, 3.2, 3.3 and 3.3 to rewrite the RPO constraint into a disjunction of solved problems. Check the satisfiability of each solved problem by recursively applying Lemma 3.2.4, Lemma 3.2.5 or Lemma 3.2.6 until one solved problem is found satisfiable or all solved problems have been found unsatisfiable. The recursive check for solved problems terminates, because the number of variables in the recursively called problems decreases strictly.

## 3.3 Related Work

As already stated, the decidability of the satisfiability of LPO constraints was first proved by H. Comon in [Com90a]. J.P. Jouannaud and M. Okada extended this result to RPO constraints in [JO91]. A. Rubio and R. Nieuwenhuis introduced a more practicable and efficient algorithm for LPO constraints in [RN91]. Ch. Weidenbach extended their result to RPO constraints and unrestricted precedences in [Wei94]. His result is extended to arbitrary arity for multiset function symbols and has been implemented in this work.

Now we will give a (very concise) overview over the methods used in the above mentioned works:

All four algorithms start out with a set of rewrite rules to transform the input constraint into a disjunction of solved problems. From this point [Com90a] and [JO91] reduce the satisfiability of the ordering constraint to the satisfiability of simple systems and then to the satisfiability of natural simple systems (which is easy to decide). However, this process is very complicated and adds another degree of complexity. Also, their descriptions of the algorithms are quite vague.

The algorithm in [RN91] introduces a nice trick: suppose the input constraint has been transformed into a disjunction of solved problems. As we saw above, left atoms and equations can be disregarded, as they play no role for the satisfiability. Hence, the object is to transform a solved problem $P$ into another solved problem $R$ with

$$P \text{ satisfiable } \Leftrightarrow R \text{ satisfiable and } R \text{ has less right atoms than } P$$

The variable $x_i$ in the right atom $x_i \succ t_i$ can be replaced by $\mathrm{succ}(t_i)$ if $t_i$ is the maximal right term. As it is not known which term $t_i$ is maximal, we have to guess and include $t_i \succeq t_j \mid \forall j$ in $R$. Also, succ is not total on ground terms for arbitrary precedences. This problem is solved in [RN91] by restricting the precedence to *simple precedence*, where succ is total:

$$succ_{\mathcal{F}}(t\sigma) = f(\vec{0}, t\sigma) \quad \text{if } \mathcal{F} \text{ simple}$$

Now this step is recursively applied to $R$, the number of right atoms decreases in each step and hence the satisfiability can be decided.

For RPO constraints succ is not total, even for simple precedences. So [Wei94] introduces another step to the above procedure: if $\mathrm{succ}(t_i)$ is defined, proceed as above. If $\mathrm{succ}(t_i)$ is not defined, replace the maximal right atom in $P$ with $x_i \simeq t_i$, check the satisfiability of the new problem. If it is satisfiable with ground substitution $\sigma$, replace the maximal right atom in $P$ with $x_i \simeq \mathrm{succ}(t_i\sigma)$ (as RPO is total on ground terms, succ is total on ground terms, too). Then go on as in [RN91].

This does not only enable us to apply the methods in [RN91] to RPO constraints, but also allows arbitrary precedences, as the need for a total successor function is circumvented.

# 4

# Implementation

In the previous chapter we proved that the satisfiability of RPO constraints is decidable and also gave the ideas for an actual implementation. In this chapter we will give an overview over the implementation, look at problems and how they are solved, discuss performance issues and look at some details where the implementation is not straightforward.

## 4.1 Notation

For the presentation of algorithms we use a standardized notation: types like **integer** or **constraint** and keywords like **algorithm**, **begin**, **end**, **if**, or **then** are written in **boldface**. The scope of conditional and loop constructs ends with an **fi** or **od** (for **if** and **do** respectively) and is marked by indentation.

The keywords for conditional constructs (**if**, **then**, **else**, **elseif** and **fi**) and loop constructs (**while**, **forall**, **do** and **od**) have the usual meaning (as in `C`, `C++` or other imperative programming languages). Two keywords may require further explanation: **break** causes termination of the smallest enclosing loop statement, and **continue** transfers control to the end of the smallest enclosing loop construct.

Comments follow `C++` syntax: `//` denotes the beginning of a comment, which ends at the end of the line.

## 4.2 Technical Details

The implementation was done in `C++`, and a library with standard implementations of data structures and methods for symbols, signatures, terms and formulas was used. In this library called `EARL`, terms and formulas are implemented in the usual way: they are stored as trees where the nodes contain a symbol and a (possibly empty) list of pointers to subterms or subformulas respectively. (See [WMCK95]).

## 4.3 Overview

R. Nieuwenhuis proved in [Nie93b] that deciding the satisfiability of LPO–constraints is NP–hard and the same proof applies for RPO–constraints (as LPO is contained in RPO). This implies that a careful implementation and additional simplifications of the problem are crucial for the performance and hence the usefulness in real applications.

A very performance critical part of the algorithm is the rewrite system. We saw that it rewrites the input constraint into a disjunction of solved problems. Hence for satisfiable constraints it suffices to find *one* satisfiable solved problem. So a major improvement compared to the approach to apply all rules exhaustively is to compute the solved problems *incrementally*. This saves a lot of computations, as some rules produce *many* new disjunction elements: for instance, the *Decomposition-mul* rule rewrites an inequation $f(s_1, \ldots, s_n) \# f(t_1, \ldots, t_m)$ into $(n! \cdot m! \cdot \min(n, m))$ new problems.

Figure 4.1 shows the main loop of the algorithm. All solved problems $P$ of a constraint $T$ are computed in function `GIVE_SOLVED_PROBLEM` and then checked by the recursive algorithm suggested by the Lemmata 3.2.4, 3.2.5 and 3.2.6. If no solved problem was found satisfiable, $\perp$ is returned, otherwise $\top$ or one solution for $T$. The procedure `GIVE_SOLVED_PROBLEM` is by far the largest and most complicated part of the implementation: it includes the application of the rewrite rules with additional methods to compute the solved problems incrementally and also a simplifier for RPO inequations.

## 4.4 Computing the Successor of a Term

In lines 20 and 35 of Figure 4.1 we need to compute the successor of a term. Recall Definition 3.2.4: for the definition of the successor of a ground term $s = g(s_1, \ldots, s_n) \in \mathcal{T}(\mathcal{F})$ and $f = g$, $\mathrm{Stat}(f) = \mathrm{mul}$, we assume that the $s_i$ are sorted in descending order wrt. $\succ_{\mathrm{rpo}}$. Therefore, we actually have to sort the subterms in the implementation of the successor function.

To compare the elements, the sort algorithm (*InsertionSort*) uses the simplifier (see Section 4.8 for a description of the simplifier and *Inser-*

```
algorithm SATISFIABLE(constraint $T$)
 begin
    while true do
       $P$ :=GIVE_SOLVED_PROBLEM($T$)
       if $P = \bot$ or $P = \varnothing$ then return $\bot$ fi
       if $P = \top$ then return $\top$ fi
       if there is no right term $t_i \in P$ then return $P$ fi
       forall right atoms $x_i \# t_i$ in $P$ do
          if $\# = \succeq$ then     // see Lemma 3.2.6
             $R := P \setminus \{t'_j[x_i]_p \# x'_j \mid p \neq \lambda, \forall j\} \cup \{t_i \succeq t_j \mid \forall (x_j \succ t_j) \in P\}$
                   $\cup \{t_i \succ t_j \mid \forall (x_j \succeq t_j) \in P\} \cup \{x_i \simeq t_i\}$
             SOLUTION:=SATISFIABLE($R$)
             if SOLUTION $\neq \bot$ then
                return SOLUTION
             else
                continue
             fi
          else // $\# = \succ$
             if $t_i \in$ Dom(succ) then     // see Lemma 3.2.4
                $R := P \setminus \{t'_j[x_i]_p \# x'_j \mid p \neq \lambda, \forall j\}$
                      $\cup \{t_i \succeq t_j \mid \forall j\} \cup \{x_i \simeq \mathrm{succ}(t_i)\}$
                SOLUTION:=SATISFIABLE($R$)
                if SOLUTION $\neq \bot$ then
                   return SOLUTION
                else
                   continue
                fi
             else // $t_i \notin$ Dom(succ), see Lemma 3.2.5
                $R := P \setminus \{t'_j[x_i]_p \# x'_j \mid p \neq \lambda, \forall j\} \setminus \{x_i \# t_j \mid \forall j\}$
                      $\cup \{t_i \succeq t_j \mid \forall j\} \cup \{x_i \simeq t_i\}$
                $\sigma$ :=SATISFIABLE($R$)
                if $\sigma = \bot$ then
                   continue
                fi
                $Q := P \setminus \{t'_j[x_i]_p \# x'_j \mid p \neq \lambda, \forall j\}$
                      $\cup \{t_i \succeq t_j \mid \forall j\} \cup \{x_i \simeq \mathrm{succ}(t_i \sigma)\}$
                SOLUTION:=SATISFIABLE($Q$)
                if SOLUTION $\neq \bot$ then
                   return SOLUTION
                fi
             fi
          fi
       od
    od
 end
```

Figure 4.1: SATISFIABLE: Satisfiability Check for RPO Constraints

*tionSort*). The simplifier can be used not only to find simplifications for (in-)equations on terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$, but also to compute the relation between two ground terms.

For the other cases in the definition of the successor function, the implementation is straightforward.

## 4.5 Incremental Computation of Solved Problems

In this section we will look at the details of `GIVE_SOLVED_PROBLEM`. The input constraint $T$ is considered to be of the following form:

$$T = \bigvee_{1 \le i \le n} F_i \quad F_i \text{ formula}, n \ge 1$$

Now the constraint is stored as a list of pairs $\langle F_i, c_i \rangle$, where $c_i$ is a counter for the number of already computed new disjuncts. The counter is initially set to 0. The constructor function for **constraint** also does some preprocessing: nested "$\wedge$"'s and "$\vee$"'s are flattened, a "$\perp$" in disjunctions is silently discarded and a "$\top$" in the top-level disjunction leads immediately to the result "satisfiable". Table 4.1 shows the preprocessing rules.

$$\begin{array}{l}
\wedge(F_1, \ldots, F_{i-1}, \wedge(F'_1, \ldots, F'_n), F_{i+1}, \ldots, F_m) \\
\qquad \to \wedge(F_1, \ldots, F_{i-1}, F'_1, \ldots, F'_n, F_{i+1}, \ldots, F_m) \\
\vee(F_1, \ldots, F_{i-1}, \vee(F'_1, \ldots, F'_n), F_{i+1}, \ldots, F_m) \\
\qquad \to \vee(F_1, \ldots, F_{i-1}, F'_1, \ldots, F'_n, F_{i+1}, \ldots, F_m) \\
\vee(F_1, \ldots, F_{i-1}, \perp, F_{i+1}, \ldots, F_m) \\
\qquad \to \vee(F_1, \ldots, F_{i-1}, F_{i+1}, \ldots, F_m) \\
\vee(F_1, \ldots, F_{i-1}, \top, F_{i+1}, \ldots, F_m) \\
\qquad \to \top
\end{array}$$

Table 4.1: Preprocessing of Constraints

An outline of `GIVE_SOLVED_PROBLEM` is given in Figure 4.2. This figure shows how rules of three different types wrt. the incremental computation of solved problems are handled. It consists of one big loop, every rule is tried for applicability on the first formula, if none applies it is a solved problem. Then it is removed from the list and returned. In order to avoid recursion in this procedure, all rules are only checked top-level in the elements of the list. This requires another rule to keep the rewrite system complete: ($\neg\neg a \to a$). Also, all rules that don't have a top-level "$\wedge$" on the left side

```
algorithm GIVE_SOLVED_PROBLEM(constraint T)
// T list of pairs ⟨F_i, c_i⟩ with F_i formula and c_i integer.
begin
    while true do
        F := F_1
        // ...
        if rule_of_type_1 applies to F then
            // rule: F → F',  F' no disjunction
            replace ⟨F_1, 0⟩ with ⟨F'_1, 0⟩ in list T
            continue
        fi
        // ...
        if rule_of_type_2 applies to F then
            // rule: F → F'_1 ∨ F'_2
            remove ⟨F_1, 0⟩ from list T
            T := cons(⟨F'_1, 0⟩, cons(⟨F'_2, 0⟩, T))
            continue
        fi
        // ...
        if rule_of_type_3 applies to F then
            // rule: F → ⋁_{1≤k≤p} F'_k for p ≥ 3
            if c_1 = p − 1 then
                remove ⟨F_1, 0⟩ from T
                T := cons(⟨F'_p, 0⟩, T)
                continue
            else
                replace ⟨F_1, c_1⟩ by ⟨F_1, (c_1 + 1)⟩ in T
                T := cons(⟨F'_(c_1+1), 0⟩, T)
                continue
            fi
        fi
        // ...
        // All rule tested, none applied:
        remove ⟨F_1, 0⟩ from T
        return F
    od
end
```

Figure 4.2: GIVE_SOLVED_PROBLEM: Computation of a Solved Problem

(like $(\neg(s \succ t) \to t \succeq s)$) have to be checked for the top-level subformulas of conjunctions. So the just mentioned example is implemented as:

$$\bigwedge_{1 \le i \le n} F_i \wedge \neg(s \succ t),\, n \ge 0 \quad \to \quad \bigwedge_{1 \le i \le n} F_i \wedge t \succeq s$$

Now let us look at how the incremental computation of a solved problem is done: the idea is to compute just one new disjunction element at a time and to then apply the rewrite system to the new formula. This will eventually lead to a solved problem, which can be checked for satisfiability. If the check is not successful, compute the next disjunction element with the rule where we left off. The counter $c_i$ in $\langle F_i, c_i \rangle$ is used to remember how many new formulas have already been computed.

Three cases are distinguished: if a rule produces just one new disjunction element, there is no need for the incremental approach, hence the old formula is just replaced by the new one. This is shown as rule of type 1 in lines 7–11 of Figure 4.2. In the second case, shown in lines 13–18, two new disjunction elements are generated. Here both new formulas replace the old one, as no space would be saved by generating just one and keeping the old one. The interesting case is shown in lines 20–30: a rule which generates three or more new disjunction elements. One new formula is generated, put in front of the old one and the counter of the old formula is incremented. Then we jump to the beginning of the loop and the rewrite system is now working on the new formula. Eventually, we will get back to the old formula and generate the next new one.

Let us look at an example: suppose `GIVE_SOLVED_PROBLEM` is working on the following constraint:

$$C \;=\; \{\langle(\wedge(\vee(\neg(f(y) \succ b), \bot, b \succ a), a \succ x)), 0\rangle\}$$

The first rule that applies is a generalization of Rule 16 in Figure 3.1:

$$\left(\bigvee_{1 \le i \le n} F_i\right) \wedge F \to \bigvee_{1 \le i \le n} (F_i \wedge F)$$

This produces the following constraint:

$$C \;=\; \{\langle(\neg(f(y) \succ b) \wedge a \succ x), 0\rangle, \langle(\wedge(\vee(\neg(f(y) \succ b), \bot, b \succ a), a \succ x)), 1\rangle\}$$

Rule 11 in Figure 3.1 leads to

$$C \;=\; \{\langle(b \succeq f(y) \wedge a \succ x), 0\rangle, \langle(\wedge(\vee(\neg(f(y) \succ b), \bot, b \succ a), a \succ x)), 1\rangle\}$$

where the first element is a solved problem. It is removed and returned. The next call of `GIVE_SOLVED_PROBLEM` looks like this:

$$\begin{aligned} C \;&=\; \{\langle(\wedge(\vee(\neg(f(y) \succ b), \bot, b \succ a), a \succ x)), 1\rangle\} \\ \to C \;&=\; \{\langle(\bot \wedge a \succ x), 0\rangle, \langle(\wedge(\vee(\neg(f(y) \succ b), \bot, b \succ a), a \succ x)), 2\rangle\} \\ \to C \;&=\; \{\langle \bot, 0\rangle, \langle(\wedge(\vee(\neg(f(y) \succ b), \bot, b \succ a), a \succ x)), 2\rangle\} \end{aligned}$$

Here the first element $\bot$ is immediately discarded as unsatisfiable and the last step is done:

$$\begin{array}{lcl} \rightarrow C & = & \{\langle(\wedge(\vee(\neg(f(y) \succ b), \bot, b \succ a), a \succ x)), 2\rangle\} \\ \rightarrow C & = & \{\langle(b \succ a \wedge a \succ x), 0\rangle\} \end{array}$$

The last solved problem is returned and the remaining constraint is empty.

This example uses the most simple rule for the incremental approach: it is straightforward to compute the $i$-th new formula. For other rules, especially the rules dealing with multiset status function symbols, this is quite complicated. We have to deal with problems like the computation of the $i$-th permutation of $(1, \ldots, n)$ and case differentiations for variable arity function symbols.

It should be noted that this method to compute the solved problems incrementally relies on the way the rules are checked and applied. All rules are checked and applied on the first element in the list in a fixed order. After a rule has been applied, we start at the beginning. This ensures that for a first element $\langle F_i, i\rangle$ with $i > 0$ exactly the rule that has previously been applied incompletely will be applied.

Another important aspect for the order of the rewrite rules is performance: it is easy to see that rules which make the constraint smaller and can be checked fast should be tried first.

In the implementation, the normalization rules in Table 3.1 are tested first: we start with Rules 1–12, as they reduce the size of the problem. Then the simplifier described in Section 4.8 is called on equations and inequations. And finally the rest of the rules in Table 3.1 is checked. As noted before, we also have to check the rules inside conjunctions, hence now rules 1, 2 and 9–15 and the simplifier are checked for the top level subformulas of a disjunction.

At this point, the head of the constraint list has the following form:

$$\bigwedge_{1 \leq j \leq n} F_i \text{ with } F_i \in \{\bot, \top, s_i \succ t_i, s_i \succeq t_i, s_i \simeq t_i\}$$

Note that $F_i \in \{\bot, \top\}$ only if $n = 1$. As mentioned before, a $\bot$ would be discarded and a $\top$ would be returned. Next all rules in Table 3.2 are checked, then the rules in Tables 3.3 and 3.4. Again, the simplifying rules are checked earlier and rules that increase the size of the problem are checked later. One exception is the *Simplification* rule in Table 3.3: as the condition "$(x \simeq t \wedge P_{\simeq})$ is solved" is complicated to check, we just test this rule last. Since all other rules have been checked before without being applicable, this condition is fulfilled and doesn't need to be checked.

## 4.6 Fast Computation of Permutations

Let's recall the first part of Rule *Decomposition-mul* in Table 3.2 as an example:

$$f(t_1,\dots,t_n) \simeq f(s_1,\dots,s_n) \wedge P \to \bigvee_{\pi\in S^n} \left[ \left( \bigwedge_{1\le j\le n} t_j \simeq s_{\pi(j)} \right) \wedge P \right]$$
$$\text{if } \mathrm{Stat}(f) = \mathrm{mul}$$

This rule rewrites one equation into a disjunction of $n!$ new equations (for $n = \mathrm{arity}(f)$). For the incremental approach, the problem is not to compute all permutations of $(1,\dots,n)$, but to compute the $i$-th permutation of $(1,\dots,n)$ in some enumeration of all permutations. The naive way to do this, is to just enumerate all permutations up to the $i$-th permutation, but this means enumerating $1+2+\cdots+n! = n!(n!+1)/2$ permutations.

As this is very bad wrt. performance, we need an algorithm that computes every permutation only once. I spent a lot of time thinking up an algorithm that computes the $i$-th permutation of all permutations in increasing lexicographical order. My algorithm uses $O(n^2)$ copies to compute one permutation. Later I found out that D.E. Knuth introduced an algorithm in [Knu73] which computes a distinct integer number for each permutation of $(1,\dots,n)$:
Given a permutation $(U_1,\dots,U_n)$ of $(1,\dots,n)$, the algorithm computes an integer $f(U_1,\dots,U_n)$ with $0 \le f(U_1,\dots,U_n) \le n!$ and $f(U_1,\dots,U_n) = f(V_1,\dots,V_n) \Leftrightarrow (U_1,\dots,U_n) = (V_1,\dots,V_n)$.

Knuth also suggests that this algorithm can be run backwards, i.e. compute a distinct permutation for an integer. The generated permutations are not ordered in increasing lexicographic order, but that doesn't matter for our application.

(Note: There is a minor bug in Knuth's description of his algorithm: "set $s \leftarrow f \mod r$" should read "set $s \leftarrow (f \mod r) + 1$". I reported the bug to Knuth and it made its way into the errata list, now I'm waiting for my \$2.56 cheque...).

This algorithm takes only $O(n)$ copies to compute a permutation. So Knuth's algorithm is used in the final version of the implementation, see Algorithm 4.3.

## 4.7 Cycle Detection

The rules called *Cycle* in Table 3.2 and Table 3.3 have been implemented by building a graph and using a standard cycle–detection algorithm (Topological Sort, see [Meh84]). We will see now how exactly this works with the

```
algorithm PERM(int n,int i)
 int A[n]
 int r, s, f, i
 begin
    // initialize A with 0,1,...,n−1:
    forall 0 ≤ i ≤ n do
       A[i] = i
    od
    f = i
    forall 2 ≤ r ≤ n do
       s = f mod r
       f = ⌊f/r⌋
       swap A[s] and A[r−1]
    od
    return A
 end
```

Figure 4.3: PERM: Compute the $i$-th Permutation of $(0, \dots, n-1)$

first *Cycle* rule as an example:

$$x_1 \simeq t_1[x_2]_{p_1} \wedge \dots \wedge x_n \simeq t_n[x_1]_{p_n} \wedge P \rightarrow \bot$$
$$\text{if } \exists\, n, i \text{ with } 1 \leq i \leq n, \ \ p_i \neq \lambda$$

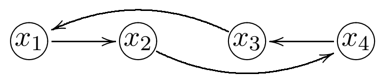
First, the equational part of the problem $P_{\simeq}$ is stored in two lists. One list contains all $x_i \simeq t_i[x_j]_{p_i}$ with $p_i \neq \lambda$, the other list all the $x_i \simeq t_i[x_j]_{p_i}$ with $p_i = \lambda$, i.e. $x_i \simeq x_j$. Then we iterate over the second list, using each equation as substitution $[x_j/x_i]$ on the equations in both lists and deleting the equation afterwards. This will eventually leave us with an empty second list.

Now the first list is used to build a graph: the $x_i$'s on the left side establish the nodes. Each $x_i \simeq t_i[x_j]_{p_i}$ establishes an edge from node $x_i$ to node $x_j$ (if there is a node $x_j$). The graph is implemented by an array of adjacency lists, which in turn are lists of integers.

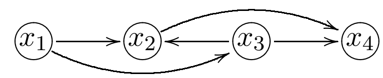
To make things clearer, we will look at two examples: a problem with cycles, shown in Problem 4.1, and a cycle–free problem, shown in Problem 4.2.

$$x_1 \simeq f(g(x_2, a), b) \wedge x_2 \simeq h(x_4) \wedge x_3 \simeq g(x_1, b) \wedge x_4 \simeq f(0, x_3) \wedge P \qquad (4.1)$$
$$x_1 \simeq f(x_2, x_3) \wedge x_2 \simeq g(h(a), x_4) \wedge x_3 \simeq f(x_2, x_4) \wedge x_4 \simeq f(a, b) \wedge P \qquad (4.2)$$

Figure 4.4 shows the corresponding graphs and the adjacency list representation of the graphs. The adjacency list representation of a graph is used as input for a standard cycle–detection algorithm: *Topological Sort*. Topological Sort is used here just to decide if a graph contains a cycle and hence the actual order of the nodes is not computed. The algorithm shown in Figure 4.5 uses an array to store the indegree of each node and a stack to store



(a) Graph for Example 4.1



(b) Graph for Example 4.2

$$\begin{array}{ll} \mathbf{x_1}: & G[0] = \{1\} \\ \mathbf{x_2}: & G[1] = \{3\} \\ \mathbf{x_3}: & G[2] = \{1\} \\ \mathbf{x_4}: & G[3] = \{2\} \end{array}$$

(c) Adjacency List for the Above Graph

$$\begin{array}{ll} \mathbf{x_1}: & G[0] = \{1, 2\} \\ \mathbf{x_2}: & G[1] = \{3\} \\ \mathbf{x_3}: & G[2] = \{1, 3\} \\ \mathbf{x_4}: & G[3] = \{\ \} \end{array}$$

(d) Adjacency List for the Above Graph

Figure 4.4: Graphs and the Corresponding Adjacency Lists

zero–indegree nodes. It works as follows: choose a node with zero indegree (i.e., no incoming edges), remove the node and its outgoing edges from the graph. Repeat until the graph is empty or no node with zero indegree is left. In the first case the graph is cycle–free, in the latter it is not.

Our first example contains no zero–indegree node and hence is not cycle–free. In the second example node $x_1$ has zero indegree and is removed with its outgoing edges, then node $x_3$, node $x_2$ and last node $x_4$. Thus the second example contains no cycle.

Cycle–Detection works very similar for the second *Cycle* rule:

$$t_1 \# s_1[t_2]_{p_1} \wedge \cdots \wedge t_n \# s_n[t_1]_{p_n} \wedge P \to \bot$$
$$\text{if some } \# = \succ \text{ or some } p_i \neq \lambda$$

For this rule, the $t_i$'s are used as nodes, and edges point from $t_k$ to $t_i$ for $t_i \# s_i[t_j]_{p_i}$ and $t_k \# s_k[t_i]_{p_k}$.

## 4.8 The Simplifier

The previous sections of this chapter described how the algorithm of Chapter 3 has been implemented and how some details have been solved. In this section we will look at the simplifier, which is not needed for the correctness of our algorithm but does a lot to improve efficiency.

For ground terms, the relation between two terms $t_1, t_2$ can easily be computed, e.g. by a simple top–down, recursive procedure, as the definition of the ordering implies. This approach is quite inefficient, as relations between the same pairs of subterms may be computed many times. Wayne Snyder shows in [Sny93] that this algorithm has a worst–case exponential complexity (in $n = |t_1| + |t_2|$). The bottom–up approach turns out to be

```
algorithm TopSort(graph G, int n)
// G is an array of n lists of integers
// If G contains cycles, TRUE is returned, FALSE otherwise.
begin
    stack<int> zeroindeg
    array<int> indeg[n]
    int count, v
    // first compute indegree of all nodes:
    forall 0 ≤ i < n do
        forall j ∈ G[i] do
            indeg[j] := indeg[j] + 1
        od
    od
    // now put zero indegree nodes on stack:
    forall 0 ≤ i < n do
        if indeg[i] = 0 then
            zeroindeg.Push(i)
        fi
    od
    // main loop:
    while zeroindeg not empty do
        remove node:
        v = zeroindeg.Pop
        count := count + 1
        // "remove" edges:
        forall i in G[v] do
            indeg[i] := indeg[i] − 1
            if indeg[i] = 0 then
                // put new zero-indegree nodes on stack:
                zeroindeg.Push(i)
            fi
        od
    od
    if count < n then
        return TRUE
    elsif
        return FALSE
    fi
end
```

Figure 4.5: `TopSort`: Cycle Check for Directed Graphs

much better: generate all subterms, sort them in increasing order wrt. their size, compare all subterms in this order and store the results. This algorithm has an $O(n^2)$, $n = |t_1| + |t_2|$ complexity. For ground terms and total precedence, Snyder introduced an even better algorithm with $O(n \log n)$ complexity in [Sny93].

For our algorithm, it was desirable to apply as many simplifications to the problem as possible, *before* the more complex rewrite rules have to be applied. There are many cases where an inequation $t_1 \# t_2$ or an equation $t_1 \simeq t_2$ can be reduced to $\bot$ (not satisfiable) or $\top$ (tautology).
Let us look at some examples, where "0" denotes the smallest constant wrt. the precedence:

$$\begin{array}{lcl} h(a,b,g(z,f(g(a,0),x))) \succ h(a,b,g(z,f(g(a,y),x))) & \rightarrow & \bot \\ h(a,b,g(z,f(g(a,c),x))) \succ h(a,b,g(z,f(g(a,0),x))) & \rightarrow & \top \end{array}$$

For both examples, the RPO *Decomposition* rules would produce lots of new equations and inequations, but comparing the terms wrt. the ordering can decide the satisfiability of the inequations fast.

I therefore implemented a simplifier which compares terms (with variables) using the bottom–up approach with $O(n^2)$ complexity. Furthermore, all computed results are stored between calls to the simplifier.

The implementation of the simplifier consists of three components:

- `SimplifierStorage`, data structure to store the already computed relations between terms

- `SimplifierInsert`, a procedure which inserts all subterms of a term into the data structure and computes the relation to all other terms in the structure

- `SimplifierLookup`, a procedure which looks up two terms, inserts them if necessary, finds out the relation between them (if known) and returns the result.

Let us examine `SimplifierLookup` first (Figure 4.6), as it is the user interface for the simplifier. It is rather simple: lines 3–5 check if $t_1$ and $t_2$ are equal. This is done by calling a procedure which checks if the terms are identical up to permutations of subterms for multiset status function symbols.
Then (in lines 6–11) both terms are looked up in the storage data structure described below, if they are already stored. If not, they are inserted with the `SimplifierInsert` procedure. In lines 12–20, the computed relation for both terms is retrieved and returned.

Now let us look at the more interesting procedure `SimplifierInsert` and the storage data structure: The data structure `SimplifierStorage` is a linked list of records. Each record holds a term and a pointer to the head of the list of pointers to terms greater than itself. An example for

**algorithm** `SimplifierLookup`(**term** $t_1, t_2$)<br>
**begin**<br>
    **if** $t_1 \simeq_{\mathrm{rpo}} t_2$ **then**<br>
        **return** 0<br>
    **fi**<br>
    **if** $t_1$ not in simplifier storage **then**<br>
        `SimplifierInsert`($t_1$)<br>
    **fi**<br>
    **if** $t_2$ not in simplifier storage **then**<br>
        `SimplifierInsert`($t_2$)<br>
    **fi**<br>
    **if** $t_1 \succ_{\mathrm{rpo}} t_2$ **then**<br>
        **return** 1<br>
    **fi**<br>
    **if** $t_2 \succ_{\mathrm{rpo}} t_1$ **then**<br>
        **return** $-1$<br>
    **fi**<br>
    **if** $t_1$ and $t_2$ not comparable **then**<br>
        **return** $-2$<br>
    **fi**<br>
**end**

Figure 4.6: `SimplifierLookup`: Look Up Relation Between Two Terms

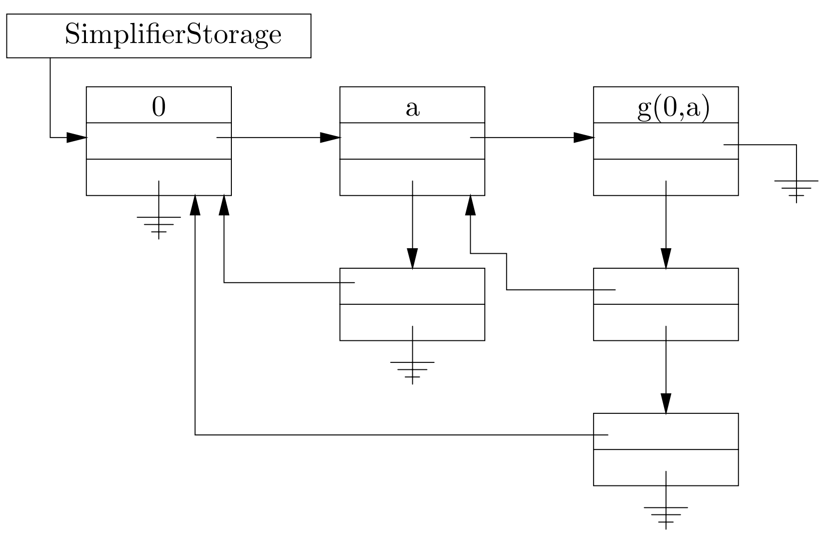



Figure 4.7: `SimplifierStorage`: The Simplifier Data Structure

`SimplifierStorage` holding the terms 0, $a$ and $g(0, a)$ is shown in Figure 4.7.

Initially, the smallest constant symbol 0 is inserted, further terms are inserted by `SimplifierInsert`, which is called only by `SimplifierLookup`. All the work is done in the procedure `SimplifierInsert`: first all subterms of the term to be inserted are generated. E.g. in the example above for $h(a, b, g(z, f(g(a, 0), x)))$ the subterms

$$\{a, b, g(z, f(g(a, 0), x)), z, f(g(a, 0), x), g(a, 0), x, a, 0\}$$

are generated. This list of terms is now sorted in increasing order wrt. their size, and multiple occurrences of a term are removed. In the example, this would result in the list

$$\{a, b, z, x, 0, g(a, 0), f(g(a, 0), x), g(z, f(g(a, 0), x))\}$$

Now the subterms are inserted in `SimplifierStorage` in this order, which guarantees that every subterm of the term to be inserted is already present in the data structure. Thus the relation to all terms already present in `SimplifierStorage` can be computed without recursion. For the following example we will represent the storage data structure as list of pairs where each pair contains a term and the list of smaller terms wrt. $\succ_{\mathrm{rpo}}$. Assume a fresh copy of the data structure, containing just the term 0, and a precedence $h \succ g \succ f \succ b \succ a \succ 0$:

$$\texttt{SimplifierStorage} = \{\langle 0, \{\}\rangle\}$$

In the example, the term $a$ is the first term to be added to the list. $a$ is compared to 0 and as $a \succ_{\mathrm{rpo}} 0$ (first case in Definition 2.3.12) a pointer to 0 is added to the list of $a$:

$$\begin{aligned}\texttt{SimplifierStorage} = \{&\langle 0, \{\}\rangle,\\ &\langle a, \{0\}\rangle\}\end{aligned}$$

Next, $b$, $z$ and $x$ are inserted, for the variables $z$ and $x$ no relation to the other terms can be computed:

$$\begin{aligned}\texttt{SimplifierStorage} = \{&\langle 0, \{\}\rangle,\\ &\langle a, \{0\}\rangle,\\ &\langle b, \{0, a\}\rangle,\\ &\langle z, \{\}\rangle,\\ &\langle x, \{\}\rangle, \}\end{aligned}$$

Now $g(a, 0)$ is inserted: it is greater than 0, $a$ and $b$ (again, first case in

Definition 2.3.12):

$$
\begin{aligned}
\texttt{SimplifierStorage} = \{ & \langle 0, \{\} \rangle, \\
& \langle a, \{0\} \rangle, \\
& \langle b, \{0, a\} \rangle, \\
& \langle z, \{\} \rangle, \\
& \langle x, \{\} \rangle, \\
& \langle g(a,0), \{0, a, b\} \rangle \}
\end{aligned}
$$

Next, $f(g(a,0),x)$ is inserted. It is easy to see, that it is greater than 0, $a$, $b$ and $x$ but not comparable with $z$. Now the algorithm has to check if $f(g(a,0),x) \succ_{\mathrm{rpo}} g(a,0)$. As $g \succ f$, the subterms of $f(g(a,0),x)$ must be checked against $g(a,0)$ (according to Definition 2.3.12, Case 2). All these relations have been computed at this point, hence no recursion is needed. So $f(g(a,0),x) \succ_{\mathrm{rpo}} g(a,0)$ is derived and stored:

$$
\begin{aligned}
\texttt{SimplifierStorage} = \{ & \langle 0, \{\} \rangle, \\
& \langle a, \{0\} \rangle, \\
& \langle b, \{0, a\} \rangle, \\
& \langle z, \{\} \rangle, \\
& \langle x, \{\} \rangle, \\
& \langle g(a,0), \{0, a, b\} \rangle, \\
& \langle f(g(a,0),x), \{0, a, b, x, g(a,0)\} \rangle \}
\end{aligned}
$$

Here is the data structure after inserting $g(z, f(g(a,0),x))$:

$$
\begin{aligned}
\texttt{SimplifierStorage} = \{ & \langle 0, \{\} \rangle, \\
& \langle a, \{0\} \rangle, \\
& \langle b, \{0, a\} \rangle, \\
& \langle z, \{\} \rangle, \\
& \langle x, \{\} \rangle, \\
& \langle g(a,0), \{0, a, b\} \rangle, \\
& \langle f(g(a,0),x), \{0, a, b, x, g(a,0)\} \rangle, \\
& \langle g(z, f(g(a,0),x)), \\
& \quad \{0, a, b, z, x, g(a,0), f(g(a,0),x)\} \rangle \}
\end{aligned}
$$

Finally, after inserting $h(a, b, g(z, f(g(a, 0), x)))$:

$$
\begin{aligned}
\texttt{SimplifierStorage} = \{ & \langle 0, \{\}\rangle, \\
& \langle a, \{0\}\rangle, \\
& \langle b, \{0, a\}\rangle, \\
& \langle z, \{\}\rangle, \\
& \langle x, \{\}\rangle, \\
& \langle g(a, 0), \{0, a, b\}\rangle, \\
& \langle f(g(a, 0), x), \{0, a, b, x, g(a, 0)\}\rangle, \\
& \langle g(z, f(g(a, 0), x)), \\
& \quad \{0, a, b, z, x, g(a, 0), f(g(a, 0), x)\}\rangle, \\
& \langle h(a, b, g(z, f(g(a, 0), x))), \\
& \quad \{0, a, b, z, x, g(a, 0), f(g(a, 0), x), \\
& \quad\; g(z, f(g(a, 0), x))\}\rangle\}
\end{aligned}
$$

Now we will describe the `SimplifierInsert` procedure in a more formal way, as the example doesn't cover all details of it. The algorithm is shown in Figure 4.8.

The input for `SimplifierInsert` is a term $t$, and the data structure `SimplifierStorage` is assumed to be a global variable (the real implementation was done in `C++` and there `insert` is a method of the class `SimplifierStorage`). First, all subterms of $t$ are generated (as in the example above) and stored in a list $\mathcal{L}$ (line 3). In line 4 list $\mathcal{L}$ is sorted wrt. the size of its elements. This is done by another procedure, which uses *Insertion Sort*, for a description see for instance [Seg92]. The complexity of Insertion Sort is $O(n^2/4)$ for the general case, but only $O(n)$ for "almost sorted" inputs. In line 5 multiple occurrences of elements are reduced to single occurrences.

Now starts the main loop over all elements in $\mathcal{L}$ in ascending order: if an element $s$ is not already present in `SimplifierStorage`, it is added (lines 7 and 8), otherwise the next $s$ is processed (line 9). Lines 10 and 11: if $s$ is a variable, then it is incomparable to all terms already in the data structure: if some $t[s]_p$ were in the data structure, then (due to the order in which terms are inserted) $s$ would be already in the data structure, too.

In Line 12 starts the inner loop over all terms $u$ stored in `SimplifierStorage`: if $u$ is a variable it is comparable with $s$ only if it is a subterm of $s$, see lines 13–18. Remark: if a relation is found, it is inserted into the data structure by adding a pointer to the list of the smaller term, this is noted in line 15 but omitted for the rest of the description of the algorithm.

In line 20 starts the check according to the definition of RPO: cases 1 and 2 in lines 20–34, case 3 in lines 35–37 and Figure 4.9 and case 4 in lines 38–40 and Figure 4.10.

```
algorithm SimplifierInsert(term $t$)
 begin
    generate all subterms of $t$, put them in list $L$
    sort_by_size($L$)
    remove_redundant_elements($L$)
    forall $s \in L$ do // in ascending order
       if $s \notin$ SimplifierStorage then
          add $s$ to SimplifierStorage
       else continue fi
       if $s \in \mathcal{X}$ then
          continue ; fi
       forall $u \in$ SimplifierStorage do
          if $u \in \mathcal{X}$ then
             if $u$ subterm of $s$ then
                $s \succ_{\text{rpo}} u$: add pointer to $u$ to $s$'s "greater–as" list
             fi
             continue
          fi
          // $s \notin \mathcal{X} \wedge u \notin \mathcal{X} \Rightarrow s = f(s_1, \ldots, s_m), u = g(u_1, \ldots, u_n)$
          if $f \succ g$ then // RPO Case 1
             if $\forall u_j$ : SimplifierLookup($s, u_j$) $= 1$ then
                $s \succ_{\text{rpo}} u$ ; continue ; fi
             if $\exists u_j$ : SimplifierLookup($u_j, s$) $\geq 0$ then
                $u \succ_{\text{rpo}} s$ ; continue ; fi
             continue
          fi
          if $g \succ f$ then // RPO Case 2
             if $\exists s_i$ : SimplifierLookup($s_i, u$) $\geq 0$ then
                $s \succ_{\text{rpo}} u$ ; continue ; fi
             if $\forall s_i$ : SimplifierLookup($s_i, u$) $= -1$ then
                $u \succ_{\text{rpo}} s$ ; continue ; fi
             // Relation not known:
             continue
          fi
          if $f \simeq g \wedge \text{Stat}(f) = \text{lex}$ then // RPO Case 3
             // see Figure 4.9
          fi
          if $f \simeq g \wedge \text{Stat}(f) = \text{mul}$ then // RPO Case 4
             // see Figure 4.10
          fi
       od
    od
 end
```

Figure 4.8: `SimplifierInsert`: Insert a Term in `SimplifierStorage`

```
// This fragment handles RPO Case 3 in SimplifierInsert,
// Figure 4.8, line 36
greater := smaller := unknown := false
forall 1 ≤ l ≤ m do
    if SimplifierLookup(s_l, u_l) = 0 then
        continue
    fi
    if SimplifierLookup(s_l, u_l) = 1 then
        greater := true
        break
    fi
    if SimplifierLookup(s_l, u_l) = −1 then
        smaller := true
        break
    fi
    if SimplifierLookup(s_l, u_l) = −2 then
        unknown := true
        break
    fi
od
if greater = true then
    if ∀u_j : SimplifierLookup(s, u_j) = 1 then
        s ≻_rpo u ; continue
    elsif ∃u_j : SimplifierLookup(u_j, s) ≥ 0 then
        u ≻_rpo s ; continue
    fi
fi
if smaller = true then
    if ∀s_i : SimplifierLookup(s_i, u) = −1 then
        u ≻_rpo s ; continue
    elsif ∃s_i : SimplifierLookup(s_i, u) ≥ 0 then
        s ≻_rpo u ; continue
    fi
fi
if unknown = true then
    if ∃s_i : SimplifierLookup(s_i, u) ≥ 0 then
        u ≻_rpo s ; continue
    fi
    if ∃u_j : SimplifierLookup(u_j, s) ≥ 0 then
        s ≻_rpo u ; continue
    fi
fi
```

Figure 4.9: `SimplifierInsert`: Code Fragment for RPO Case 3

```
// This fragment handles RPO Case 4 in SimplifierInsert,
// Figure 4.8, line 39
list $\mathcal{S} := \{s_1, \ldots, s_m\}$ ; list $\mathcal{U} := \{u_1, \ldots, u_n\}$
forall $s_i \simeq_{\text{rpo}} u_j, \quad s_i \in \mathcal{S}, u_j \in \mathcal{U}$ do
    $\mathcal{S} := \mathcal{S} \setminus \{s_i\}$ ; $\mathcal{U} := \mathcal{U} \setminus \{u_j\}$ ; od
// first check if $s \succ_{\text{rpo}} u$:
forall $s_i \in \mathcal{S}$ do
    removed_one := false
    forall $u_j \in \mathcal{U}$ do
        if SimplifierLookup($s_i, u_j$) = 1 then
            $\mathcal{U} := \mathcal{U} \setminus \{u_j\}$
            removed_one := true
        fi
    od
    if removed_one then
        $\mathcal{S} := \mathcal{S} \setminus \{s_i\}$
    fi
od
if $\mathcal{U} = \varnothing$ then
    $s \succ_{\text{rpo}} u$
    continue
fi
// generate fresh copies of $\mathcal{S}$ and $\mathcal{U}$:
list $\mathcal{S} := \{s_1, \ldots, s_m\}$ ; list $\mathcal{U} := \{u_1, \ldots, u_n\}$
forall $s_i \simeq_{\text{rpo}} u_j, \quad s_i \in \mathcal{S}, u_j \in \mathcal{U}$ do
    $\mathcal{S} := \mathcal{S} \setminus \{s_i\}$ ; $\mathcal{U} := \mathcal{U} \setminus \{u_j\}$ ; od
// now check if $u \succ_{\text{rpo}} s$:
forall $u_j \in \mathcal{U}$ do
    removed_one := false
    forall $s_i \in \mathcal{S}$ do
        if SimplifierLookup($u_j, s_i$) = 1 then
            $\mathcal{S} := \mathcal{S} \setminus \{s_i\}$
            removed_one := true
        fi
    od
    if removed_one then
        $\mathcal{U} := \mathcal{U} \setminus \{u_j\}$
    fi
od
if $\mathcal{S} = \varnothing$ then
    $u \succ_{\text{rpo}} s$
    continue
fi
```

Figure 4.10: `SimplifierInsert`: Code Fragment for RPO Case 4

The implementation of RPO Case 3 in Figure 4.9 is self–explaining: the first loop compares the subterms on both sides in lexicographic order , then in lines 21–42 the subterm property is checked for each case. The tricky part is to cover all cases where subterms are incomparable.

And finally, Figure 4.10 shows the implementation of RPO Case 4: the toplevel subterms of $s$ and $u$ are stored in two lists $\mathcal{S}$ and $\mathcal{U}$ (line 3). Then pairs $s_i \simeq u_j,\ \ s_i \in \mathcal{S},\ u_j \in \mathcal{U}$ are removed from the lists (lines 4 and 5). In line 7–18, terms are removed from both lists (see Definition 2.3.8 for the definition of $\succ^{\mathrm{mul}}$): all terms $u_j : s_i \succ_{\mathrm{rpo}} u_j$ for some $s_i$ are removed from list $\mathcal{U}$ and then $s_i$ is removed from list $\mathcal{S}$. If list $\mathcal{U}$ is empty after the loop, then $s \succ_{\mathrm{rpo}} u$. Lines 23–43 do the same for the other direction to check if $u \succ_{\mathrm{rpo}} s$.

### 4.8.1 Experiments

In order to test the performance of the simplifier, I carried out some simple tests: Let $\mathcal{F} = \{0, a, b, c, f, g, h\}$, where $a$, $b$ and $c$ are constants, $f$ has arbitrary arity and multiset status, $h$ and $g$ have lexicographic status, $\mathrm{arity}(h) = 3$ and $\mathrm{arity}(g) = 2$, the precedence $h > g > f > c > b > a > 0$. The constraint

$$f(x, g(y, 0), y, b, h(c, a, z)) \succ f(y, a, h(b, z, a), x, g(y, 0), b) \\ \wedge x \succ h(y, 0, a)$$

is satisfiable. The constraint solver *with* the simplifier finds the solution $\{x = f(h(0, 0, a), y = 0\}$ in 0.27 CPU seconds. On the same machine (Sun IPX) the constraint solver *without* the simplifier finds the solution $\{x = h(b, a, a), y = b, z = a\}$ in 45.77 CPU seconds. Both solutions are correct, in the first case $z$ can be chosen arbitrarily. Another variation of the example above with the same signature and precedence is

$$f(x, g(y, 0), y, b, h(c, a, z)) \succ f(y, a, h(b, z, a), x, g(y, 0), b)$$

The constraint solver with the simplifier reduces the constraint to $\top$ in 0.19 CPU seconds, without the simplifier it takes 33.38 CPU seconds to find the solution $\{x = g(h(b, 0, a), 0), y = h(b, 0, a), z = 0\}$.

The simplifier does not always improve the performance that much, but at least the overhead of the simplifier doesn't slow down the constraint solver significantly in cases where it doesn't help (about 4%).

## 4.9 Ideas for Performance Improvements

The current implementation probably leaves plenty of room for experiments to improve the performance. Profiling the program revealed that the major part of the computing time is used for term copies and term comparisons. Thus, two approaches to improve performance are feasible:

1. make the term copy and term comparison operations faster

2. reduce the number of those operations

The first approach goes deep down to the implementation details and involves substituting pointer operations for term copy operations where possible, fine tuning the data structures etc. This is not of much interest here, as it can improve performance only by a constant factor and is not a special problem for this algorithm.

The second approach is more interesting: one candidate to reduce the numbers of term comparisons is the `SimplifierStorage` data structure. The current implementation uses just a linked list to store terms and hence needs $O(n)$ term comparison operations to find a term. Replacing the list by some well known search data structure as binary trees would reduce the number of compare operations to $O(\log n)$, but this would require defining a total ordering on the representation of the terms. This can be done, it is not straightforward, though: terms with multiset status function symbols require a normal form wrt. this ordering — otherwise terms equal wrt. RPO would be inserted more than once. So only an actual implementation can show if a better search algorithm will outweight the added overhead.

Another field for possible improvements is the order in which the rewrite rules are applied. In Section 4.5 we have seen the motivations for the application order used in the implementation: rules reducing the size of the problem first, rules increasing the size of the problem later. For rules equivalent in this sense, those with simpler (or no) precondition are checked first. This leaves not much room for variations: for a given formula for instance at most one of the *Decomposition* rules can be applied and hence it does not matter which one is tried first. Opposed to that, for simplifying rules like the *Cycle* rules and *Merge* the performance critical part is the precondition check. For these rules experiments with the application order may be worthwhile. The same holds for the simplifier: in the implementation it is called very early, a later execution might improve the performance. These considerations depend on the nature of the input constraints and hence the application for the algorithm.

Another possible experiment is to replace the algorithm used to sort terms wrt. their size (Insertion Sort) with Quick Sort, Heap Sort or another $O(n \log n)$ algorithm. Insertion Sort was chosen, because it is easy to implement and it benefits from almost sorted data.

The algorithms chosen for cycle detection in graphs and permutation generation are the best known algorithms for their purposes.

# Bibliography


[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[CNR95] Hubert Comon, Robert Nieuwenhuis, and Albert Rubio. Orderings, AC-theories and symbolic constraint solving. In Dexter Kozen, editor, *Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 375–385, San Diego, CA, June 1995. IEEE Comp. Soc. Press.

[Com90a] Hubert Comon. Solving inequations in term algebras. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, LICS'90*, pages 62–69. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990.

[Com90b] Hubert Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Science*, 1(4):387–411, 1990.

[Com91] Hubert Comon. Disunification: a survey. In Jean-Louis Lassez and Gordon Plötkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 322–359. MIT Press, 1991.

[Der87] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1):69–115, July 1987.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 6, pages 243–320. Elsevier Science Publishers B.V., Amsterdam, New York, Oxford, Tokyo, 1990.

[Fit90] Melvin Fitting. *First–Order Logic and Automated Theorem Proving*. Texts and Monographs in Computer Science. Springer–Verlag, 1990.

[JO91] Jean-Pierre Jouannaud and Mitsuhiro Okada. Satisfiability of systems of ordinal notations with the subterm property is decidable. In *Automata, Languages and Programming, 18th International Colloquium*, volume 510 of *LNCS*, pages 455–468, Madrid, July 1991. Springer.

[KKR90] Claude Kirchner, Hélène Kirchner, and Michaël Rusinowitch. Deduction with symbolic constraints. *Revue d'Intelligence Artificielle*, 4(3):9–52, 1990.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 2, section 3.3.2, page 64. Addison-Wesley, Reading, Massachusetts, second edition, October 1973.

[Meh84] Kurt Mehlhorn. *Data structures and algorithms. Vol. 2. Graph algorithms and NP-completeness*, chapter 1, pages 4–7. EATCS monographs on theoretical computer science. Springer, Berlin, 1984.

[Nie93a] Robert Nieuwenhuis. A new ordering constraint solving method and its applications. Technical Report MPI-I-92-238, Max-Planck-Institut für Informatik, February 1993.

[Nie93b] Robert Nieuwenhuis. Simple LPO constraint solving methods. *Information Processing Letters*, 47(2):65–69, August 1993.

[NR92] Robert Nieuwenhuis and Albert Rubio. Theorem proving with ordering constrained clauses. In *11th International Conference on Automated Deduction, CADE-11*, volume 607 of *LNCS*, pages 477–491. Springer, 1992.

[RN91] Albert Rubio and Robert Nieuwenhuis. Handling ordering constraints of clauses in theorem proving. Technical report, Research Report LSI-UPC, 1991.

[RN93] Albert Rubio and Robert Nieuwenhuis. A precedence-based total AC-compatible ordering. In *Proceedings 5th International Conference on Rewriting Techniques and Applications*, volume 690 of *LNCS*, pages 374–388. Springer, 1993.

[Rub94a] Albert Rubio. *Automated Deduction with Constraint Clauses.* PhD thesis, Departament de Llenguatges i Sistemes Informàtics de la Universitat Politècnica de Catalunya, Barcelona, April 1994. Chapter 3.2: Ordering Constraint Solvers.

[Rub94b] Albert Rubio. *Automated Deduction with Constraint Clauses.* PhD thesis, Departament de Llenguatges i Sistemes Informàtics de la Universitat Politècnica de Catalunya, Barcelona, April 1994. Chapter 2.6: The AC-RPO ordering.

[Seg92] Robert Segdewick. *Algorithmen in C++*, chapter 8: Elementare Sortierverfahren, pages 127–129. Addison-Wesley, 1992.

[Sie89] Jörg Siekmann. Unification theory. *Journal of Symbolic Computation, Special Issue on Unification*, 7:207–274, 1989.

[Sny93] Wayne Snyder. On the complexity of recursive path orderings. *Information Processing Letters*, 46:257–262, July 1993.

[Wei94] Christoph Weidenbach. An algorithm for testing satisfiability of RPO constraints. Unpublished, Private Communication, December 1994.

[WMCK95] Christoph Weidenbach, Christoph Meyer, Christian Cohrs, and Enno Keen. The EARL C++ library for automated theorem proving. Max-Planck-Institut für Informatik, 1995.