FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Proving Termination of Rewriting with the Dependency Pair Framework

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science
Computational Logic MSc

im Rahmen des Studiums

Computational Logic

eingereicht von

Milka Hutagalung
Matrikelnummer 0928046

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao. Univ. Prof. Dr. Bernhard Gramlich

Wien, 26.08.2010

(Unterschrift Verfasserin)

(Unterschrift Betreuer)

Abstract

Termination is besides confluence the most important and fundamental property of term rewriting systems (TRSs). If a TRS is terminating, then this guarantees that every strategy of computing with the system will always end with *normal forms*. But unfortunately, the undecidability of termination shows that there cannot be an automated method that will succeed in proving termination of all terminating TRSs. Numerous syntactical as well as semantical methods have been developed, but there are still many important TRSs whose termination cannot be shown. The *dependency pair framework* (introduced in [AG00] and generally formulated in [GTS05], shortly called the DP framework) is currently the most powerful method that can increase significantly the class of systems where termination is provable automatically. The power of the framework lies in its high flexibility, due to its general and modular structure, and the ability to incorporate other general termination proof methods. This thesis will summarize the DP framework and its ingredients. First, we summarize the theoretical foundations of the framework, that allows various *dependency pair processors* (DP processors) to successively simplify the initial termination proof problems. Then, as the main part of the summary, we will give a comprehensive survey of some important types of DP processors. The DP processors can be seen as the termination techniques in the DP framework. All termination proofs in the DP framework are basically obtained by repeated applications of DP processors. For each DP processor we work out the basic underlying ideas and theory, present some refinements, and illustrate their applications by examples. In addition, we will also include a short survey of some general termination proof methods, e.g. reduction orders (polynomial interpretations, matrix interpretations, simplification orders), match-bounds, and semantic labeling, which can also be applied within the DP framework.

Kurzfassung

Die Termination ist neben Konfluenz die wichtigste und grundlegendste Eigenschaft der Termersetzungsysteme (TRS). Wenn ein TRS terminiert, ist es garantiert, dass jede Strategie der Verarbeitung mit dem System immer mit einer normal Form beendet. Aber, die Unentscheidbarkeit der Termination zeigt dass, es keine automatisierte Methode gibt, die diese Eigenschaft für alle Terminierende TRS beweisen kann. Zahlreiche syntaktische sowie semantische Methoden werden entwickelt, aber es gibt noch viele wichtige TRS deren Termination nicht angezeigt werden kann. Das Dependency Pair Framework (eingeführt im [AG00] und allgemein formuliert im [GTS05], kurz DP Framework genannt) ist momentan die am weitesten verbreitete Methode. Sie kann die Klasse von Systemen, deren Terminierung automatisch beweisbar ist, signifikant erhöhen. Der Vorteil des Framework liegt an der hohen Flexibilität aufgrund seiner allgemeinen und modularen Struktur, und die Fähigkeit, andere allgemeine Terminationsbeweis-Methode zu integrieren. Die vorliegende Arbeit wird das DP Framework und seine Bestandteile zusammenfassen. Zuerst fassen wir die theoretischen Grundlagen des Framework, das diverse Dependency Pair Processors (DP processors) erlaubt, welche anfängliche Terminationsbeweis-Probleme aufeinanderfolgendlich vereinfachen. Dann, als der Hauptteil der Zusammenfassung, geben wir einen umfassenden Überblick über einige wichtige DP Processor-Typen. DP Processors lassen sich als Termination-Techniken in der DP Framework verstehen. Ein Terminationsbeweis im DP Framework ist im Wesentlichen die Wiederholte Anwendunge, einer DP Processors. Für jeden DP Processor erarbeiten wir die grundlegende Ideen und Theorie, präsentieren einige Verfeinerungen, und illustrieren Anwendunge mit Beispielen. Weiterhin erarbeiten wir auch einen Überblick über einige allgemeine Terminationsbeweis-Methoden, z.B. Reduktionsordnungen (Polynominterpretation, Matrixinterpretation, Simplifikationsordnung), Match-Bounds, und Semantic Labeling, die auch in der DP Processor angewandt werden können.

Contents

CHAPTER 1

Introduction

Term rewriting systems (TRSs) are a simple yet powerful approach to model computations of functional and logic programs. The basic idea of the TRS is very simple: it replaces a term with a term by applying the so-called *rewrite rules*. After one application of the rewrite rule, it is possible to apply other rewrite rules again. Hence it is natural to ask whether for a given TRS, the rewriting steps eventually will stop or not. This property is called *termination*. We said that a TRS is terminating, if after finitely many rewrite rules applications we always reach terms, where there is no more rewrite rule can be applied (*normal form*).

Termination is one of the most important properties of TRSs. If some TRS is terminating, then this guarantees that every strategy of computing with the system will always end with some *normal forms*. And since TRSs can be used to model the computations of functional and logic programs, then it is also possible to reduce the question of termination of the programs to termination of TRSs. Indeed, some transformational approaches for logic programs and the functional programming language Haskell have been recently developed [SGST06] [GSST06]. Moreover, having the termination property will make the *confluence* problem become easier to be checked, i.e. is it always the case that if we apply different ways of rewriting that lead to two different terms, we can always reach a common term? As stated in Newman's Lemma [New42], having the termination property will reduce the confluence problem to a simpler problem called *local confluence* problem.

Hence termination is an important property of TRSs, and having automated methods to check termination is also important. Unfortunately, for a given arbitrary TRS, the question whether it is terminating or not is undecidable. There cannot be a general procedure such that for any arbitrary input of TRS, it can answer "yes" if it is terminating and "no" if it is not terminating. However, it is possible to develop some methods to show termination of some particular TRSs. But of course the undecidability shows that these methods cannot succeed in proving termination of all terminating TRSs. One of the most common methods to show termination, is by finding some *reduction order*. These orders are an important tool for proving termination of rewrite systems. Nowadays, there is a wide range of many different possible semantical or syntactical methods to construct reduction orders. In practice, most often *simplification orders*, are

used, i.e. reduction orders with the subterm property. However, many TRSs cannot be proved terminating with simplification orders.

This problem was one of the motivation for the development of a new and the powerful method called the *dependency pair framework* (DP framework). The idea of this method is originally introduced in [AG00], and formulated in [GTS05] as a general framework that can integrate and combine many other termination proof methods. The basic idea of this framework is simple yet bright: it combines some termination proof methods but does not apply them directly to a TRS so it can simplify the termination problem gradually. The idea of gradually simplifying termination problems by applying different independent techniques, increases significantly the flexibility, modularity, and the power of proving termination. Many TRSs that cannot be shown terminating with the direct methods like reduction orders and simplification orders, now can be easily handled by this framework. The power of this framework lies in its generality, that is based on a termination criterion with the notion of *dependency pairs*.

The independent techniques used in the DP framework can be used in the *dependency pair processors* (DP processors) which transform a *dependency pair problem* (DP problem) into some (hopefully simpler) DP problems. The DP framework proves termination of TRSs by first transforming them into some initial DP problems, and then successively simplifying the initial DP problem by applying the DP processors. In fact, the DP processors are the most essential components of proving termination with the DP framework, since basically the termination proof in the DP framework is nothing more but a repeated application of the DP processors. The goal of this thesis is to give a summary of the foundation and methods used to prove termination in the DP framework by giving a comprehensive survey of some important types of the DP processors. In addition, we will also give a short survey of general termination proof methods like reduction orders, match-bounds, and semantic labeling, which can also be adapted for the DP framework.

The structure of this thesis is as follows: Chapter 2 is a preliminary chapter that briefly introduces the basic notions of the abstract reduction systems (ARSs), TRSs, and tree automata. We will use all the notions from this chapter in the later chapters. Chapter 3 is a short survey of the general termination proof methods that consists of the semantical and syntactical methods to construct reduction orders, such as polynomial interpretations, matrix interpretations, and simplification orders. We will also mention some modern transformational methods like the match-bounds and semantic labeling. Chapter 4 is a comprehensive survey of the termination proof methods based on the DP framework. First, we will give the idea of the termination criterion with the dependency pairs, as the theoretical foundation of the DP framework, and then we will briefly mention the formulation of the DP framework, DP problems, and DP processors. After that, as the main part of this chapter we will cover some important types of DP processors. We work out the basic underlying ideas and theory for each processor, point out their successfulness, present some refinements, and illustrate the applications by examples. Finally, in Chapter 5, we give some conclusion and characteristics of proving termination with the DP framework.

CHAPTER 2

Preliminaries

This chapter gives some basic definitions that will be used in the later chapters. In Section 2.1 we introduce the notion of *abstract reduction systems* (ARSs), as the general form of *term rewriting systems* (TRSs). Then in Section 2.2, we introduce the main object in this thesis: TRS, and in Section 2.3 we give some basic definitions about tree automata, since some termination techniques involve basic notions from the field of tree automata.

2.1 Abstract Reduction Systems

Definition 2.1.1 (asbtract reduction systems)**.**

An *abstract reduction system* (ARS) is a pair $(A, \to)$ consisting of a set A and a binary relation $\to \subseteq A \times A$. We call the relation $\to$ with *reduction relation* or *rewrite relation*. Instead of $(a, b) \in \to$, we write $a \to b$. A *reduction sequence* in ARS $(A, \to)$ (possibly infinite) is $a_0 \to a_1 \to a_2 \to \dots$ where $a_0, a_1, a_2, \dots$ are elements of A.

Definition 2.1.2 (normal forms, termination)**.**

Let $(A, \to)$ be an ARS,

- An element $a \in A$ is a *normal form* w.r.t. $\to$ if there is no $b \in A$ such that $a \to b$.
- An element $a_0 \in A$ is *terminating* w.r.t. to $\to$ if there is no infinite reduction sequence $a_0 \to a_1 \to a_2 \to \dots$
- The reduction $\to$ is *terminating* if for every $a \in A$, a is terminating.

Definition 2.1.3.

Let $\to$ be a reduction relation. The transitive closure of $\to$ is denoted by $\overset{+}{\to}$, and the transitive and reflexive closure of $\to$ is denoted by $\overset{*}{\to}$

Definition 2.1.4 (transitive, reflexive, irreflexive, antisymmetric)**.**

Let $\to$ be a binary relation on a set A. We say that $\to$ is

- *transitive* iff for every $a, b, c \in A$, $a \to b$ and $b \to c$ implies $a \to c$.
- *reflexive* iff for every $a \in A$, $a \to a$.
- *symmetric* iff for every $a, b \in A$, $a \to b$ implies $b \to a$.
- *antisymmetric* iff for every $a, b \in A$, $a \to b$ and $b \to a$ implies $a = b$.

Definition 2.1.5 (weak orders, strict orders)**.**
A *weak order* $\geq$ is a reflexive, transitive, and antisymmetric binary relation. A *strict order* $>$ is a transitive and irreflexive binary relation.

Definition 2.1.6 (well-founded orders)**.**
Let $>$ be an order on a set A. $>$ is called *well-founded* if and only if there is no infinite descending chain $a_0 > a_1 > a_2 > \ldots$ where $a_0, a_1, a_2, \ldots$ are elements of A.

Definition 2.1.7 (lexicographic orders)**.**
Let $>_{A_i}$ be a strict order on A_i, for $i = 1, \ldots, n$. The*lexicographic order* on $A_1 \times \ldots \times A_n$, $>_{Lex}$ is defined by: $(a_1, \ldots, a_n) >_{Lex} (a'_1, \ldots, a'_n)$ if and only if there exists i, $1 \leq i \leq n$ such that $a_1 = a'_1, \ldots, a_{i-1} = a'_{i-1}$, and $a_i >_{A_i} a'_i$.

Definition 2.1.8 (multiset orders)**.**
A *multiset*over a set A is an unordered collection of elements of A, where the elements may have multiple occurrences. The set of all finite multisets over A is denoted by $\mathcal{M}(A)$. The *multiset order* on $\mathcal{M}(A)$, $>_{Mul}$ is defined by: $M_1 >_{mul} M_2$ if and only if there exist multisets $X, Y \in \mathcal{M}(A)$ such that

- $X \neq \emptyset \subseteq M_1$,
- $M_2 = (M_1 \backslash X) \cup Y$,
- For every $y \in Y$ there exists $x \in X$ such that $x > y$.

Theorem 2.1.9. *For an ARS $(A, \to)$ the reduction $\to$ is* terminating *if and only if there is a well-founded strict order $>$ on A' and a mapping $\varphi : A \to A'$ such that for every $a, b \in A$, $a \to b$ implies $\varphi(a) > \varphi(b)$.*

This theorem describes the basic idea of proving termination of ARSs, which also can be applied to some *term rewriting systems* (TRSs). Basically TRSs are ARSs, where the reduction relation is defined on the set of terms. In the following section we will give the basic definitions for TRS, by first introducing the notion of signatures and terms.

2.2 Term Rewriting Systems

Definition 2.2.1 (signatures)**.**

A *signature* is a countable set $\mathcal{F}$ of function symbols. Every function symbol $f \in \mathcal{F}$ is associated with a natural number denoting their *arity*, i.e. the number of the arguments. We have $\mathcal{F} = \bigcup_{n \geq 0} \mathcal{F}^n$, where $\mathcal{F}^n$ denotes the set of all function symbols with arity n. Function symbols with arity 0 are called *constants*.

Definition 2.2.2 (terms)**.**

Let $\mathcal{F}$ be a signature and $\mathcal{V}$ a countable infinite set of variables. The set of terms $\mathcal{T}(\mathcal{F}, \mathcal{V})$ over $\mathcal{F}$ and $\mathcal{V}$ is recursively defined as follows:

- $\mathcal{V} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V})$
- if $f \in \mathcal{F}^n$ and $t_1, t_2, \ldots \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ then $f(t_1, \ldots t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$.

Terms containing no variables are called *ground terms*, and the set of all ground terms of $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is denoted by $\mathcal{T}(\mathcal{F})$.

Definition 2.2.3 (symbols, root symbols, size)**.**

Let $\mathcal{T}(\mathcal{F}, \mathcal{V})$ be the set of terms.

- The set of all function symbols occurring in term t is denoted by $\mathcal{F}un(t)$ and the set of all variables occurring in term t is denoted by $\mathcal{V}ar(t)$.
- If $t = f(t_1, \ldots, t_n)$ then we call f as the *root symbol* of t, i.e $root(t) = f$.
- The *size* of t is denoted by $\mid t \mid$, and it is the number of all occurrences of function symbols from $\mathcal{F}$ and variables from $\mathcal{V}$ in t.
- The number of occurrences of variable x in t is denoted by $\mid t \mid_x$.

Definition 2.2.4 (positions, subterms, replacements)**.**

- $\mathcal{P}os(t)$ is the set of *positions* of term t.

$$\mathcal{P}os(t) = \begin{cases} \{\epsilon\} & \text{if } t \text{ is variable} \\ \{\epsilon\} \cup \{ip \mid 1 \leq i \leq n \text{ and } p \in \mathcal{P}os(t_i)\} & \text{if } t = f(t_1, \ldots t_n) \end{cases}$$

- $t \mid_p$ is a *subterm* of t at position p.

$$t \mid_p = \begin{cases} t & \text{if } p = \epsilon \\ t_i \mid_q & \text{if } t = f(t_1, \ldots t_n) \text{ and } p = iq \end{cases}$$

We use $\trianglerighteq$ to denote the subterm relations, and $\triangleright$ to denote the proper/strict subterm relations. Hence for a term t, we have $t \trianglerighteq t \mid_p$ for all $p \in \mathcal{P}os(t)$, and $t \triangleright t \mid_p$ for all $p \in \mathcal{P}os(t) \backslash \{\epsilon\}$. For a term $t = f(t_1, \ldots t_n)$, we call $t_1, \ldots t_n$ as the direct subterms of t.

- $\mathcal{V}Pos(t)$ is the set of *variable positions* of t.

$$\mathcal{V}Pos(t) = \{p \in \mathcal{P}os(t) \mid t\mid_p \in \mathcal{V}\}$$

- $\mathcal{F}Pos(t)$ is the set of *non-variable positions* of t.

$$\mathcal{F}Pos(t) = \{p \in \mathcal{P}os(t) \mid t\mid_p \notin \mathcal{V}\}$$

- Positions can be weakly ordered by the ordering $\geq$. We have $p \geq q$ if $p = qr$ for some position r. If $p \leq q$, we say that p is a position above q, and q is a position below p.

- The *replacement* of the subterm of t at position p with a term s is denoted by $t[s]_p$.

$$t[s]_p = \begin{cases} s & \text{if } p = \epsilon \\ f((t_1, \ldots, t_i[s]_q, \ldots t_n) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } p = iq \end{cases}$$

Definition 2.2.5 (substitutions)**.**

A *substitution* σ is a mapping $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$, such that its domain: $\mathcal{D}om(\sigma) = \{x \in V \mid \sigma(x) \neq x\}$, is finite. Substitution can be uniquely extended to a mapping from $\mathcal{T}(\mathcal{F}, \mathcal{V})$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$ by $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$. Beside $\sigma(t)$ we also write $t\sigma$, for a term t and substitution σ. Two terms s, t are *unifiable* if there exists a substitution σ such that $\sigma(s) = \sigma(t)$, we call σ as *unifier*. A unifier σ is called the *most general unifier* (*mgu*) if for every other unifier τ, there exists a substitution ρ such that $\tau(x) = \rho(\sigma(x))$ for every variable x.

Definition 2.2.6 (term rewriting systems)**.**

A *term rewriting system* (TRS) is a pair $(\mathcal{F}, \mathcal{R})$ consist of a signature $\mathcal{F}$, and a set of rewrite rules $\mathcal{R} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V}) \times \mathcal{T}(\mathcal{F}, \mathcal{V})$. A *rewrite rule* is a pair $(l, r) \in \mathcal{R}$, and denoted by $l \to r$, where $l \notin \mathcal{V}$ and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. Instead of $(\mathcal{F}, \mathcal{R})$ we also write simply only $\mathcal{R}$ when $\mathcal{F}$ is clear from the context or irrelevant. The set of the *left-hand sides* of $\mathcal{R}$ is defined by $lhs(\mathcal{R}) = \{l \mid l \to r \in \mathcal{R}\}$, and the set of the *right-hand sides* of $\mathcal{R}$ is defined by $rhs(\mathcal{R}) = \{r \mid l \to r \in \mathcal{R}\}$.

Definition 2.2.7 (linear TRS)**.**

A term is *linear* if every variable occurs at most once in it. The rewrite rule $l \to r$ is *left-linear* if l is linear; *right-linear* if r is linear; and *linear* if both l and r are linear. We can also extend this definition to TRS, i.e. a TRS is *left-linear* (*right-linear*, *linear* resp.) if all the rewrite rules are *left-linear* (*right-linear*, *linear* resp.).

Definition 2.2.8 (closed under context and substitution)**.**

Let $\to$ be a binary relation on the set of terms $\mathcal{T}(\mathcal{F}, \mathcal{V})$.

- $\to$ is *closed under substitution* iff $s \to t$ implies $\sigma(s) \to \sigma(t)$ for all $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and substitutions σ.

- $\to$ is *closed under context* iff $s \to s'$ implies $t[s]_p \to t[s']_p$ for all terms $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and positions $p \in \mathcal{P}os(t)$.

Definition 2.2.9 (rewrite relation induced by TRS)**.**

The *rewrite relation* or *reduction relation induced by a TRS* $\mathcal{R}$ over signature $\mathcal{F}$ is denoted by $\xrightarrow[\mathcal{R}]{}$ and defined as follows: for $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, $s \xrightarrow[\mathcal{R}]{} t$ iff there exists a rewrite rule $l \to r \in \mathcal{R}$, a substitution σ, and a position $p \in \mathcal{P}os(s)$, such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$. We call $s \xrightarrow[\mathcal{R}]{} t$ a *rewrite step* or *reduction step*. In order to point out the position p where the reduction take place, we can also use the notation $s \xrightarrow[\mathcal{R}]{p} t$.

Theorem 2.2.10 (undecidability of termination)**.** *The following problem is in general undecidable. Given: A finite TRS $\mathcal{R}$. Question: Is $\mathcal{R}$ terminating or not?*

Proof. By reduction of the uniform halting problem for Turing machines [HL78]. □

2.3 Tree automata

Definition 2.3.1 (tree automata)**.**

A (finite bottom-up) *tree automaton* is a quadruple $\mathcal{A} = (\mathcal{F}, Q, Q_f, \triangle)$ consisting of a finite signature $\mathcal{F}$, a finite set Q of states disjoint from $\mathcal{F}$, a set of final states $Q_f \subseteq Q$, and a set of transition rules $\triangle$, where every transition rules has the form $f(q_1, \ldots, q_n) \to q$ with $f \in \mathcal{F}$ and $q, q_1, \ldots, q_n \in Q$.

Definition 2.3.2 (the language accepted by tree automata)**.**

A term t over a signature $\mathcal{F}$ is *accepted* by an automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \triangle)$, if $t \xrightarrow[\mathcal{A}]{*} q$ for some final state $q \in Q_f$. Here, the relation $\xrightarrow[\mathcal{A}]{}$ is the induced rewrite relation on the set of ground terms $\mathcal{T}(\mathcal{F} \cup Q)$. The set of all such terms is called the *language accepted* by $\mathcal{A}$, and denoted by $L(\mathcal{A})$.

Definition 2.3.3 (regular tree language)**.**

A *regular tree language* is a set of ground terms $L \subseteq \mathcal{T}(\mathcal{F})$, such that we can find a tree automaton $\mathcal{A}$, where the accepted language of $\mathcal{A}$, is equivalent to L, i.e. $L = L(\mathcal{A})$.

Definition 2.3.4.

Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$ and $L = \mathcal{T}(\mathcal{F})$.

- The set $(\xrightarrow[\mathcal{R}]{*})[L]$ denotes the set of ancestors of L, i.e. consists of all terms $s \in \mathcal{T}(\mathcal{F})$ such that $s \xrightarrow[\mathcal{R}]{*} t$ for some term $t \in L$, $(\xrightarrow[\mathcal{R}]{*})[L] = \{s \in \mathcal{T}(\mathcal{F}) \mid s \xrightarrow[\mathcal{R}]{*} t\}$.
- The set $(\xrightarrow[\mathcal{R}]{*})(L)$ denotes the set of descendants of L, i.e. consists of all terms $t \in \mathcal{T}(\mathcal{F})$ such that $s \xrightarrow[\mathcal{R}]{*} t$ for some term $s \in L$, $(\xrightarrow[\mathcal{R}]{*})(L) = \{t \in \mathcal{T}(\mathcal{F}) \mid s \xrightarrow[\mathcal{R}]{*} t\}$.

All the definitions and the notions about ARS, TRS, and tree automata that are mentioned here, will be used in the later chapters.

CHAPTER 3

General Termination Proof Methods

From the basic definition of ARSs and TRSs, we said that a TRS $\mathcal{R}$ is terminating if there is no infinite reduction sequence:

$$t_1 \xrightarrow[\mathcal{R}]{} t_2 \xrightarrow[\mathcal{R}]{} t_3 \xrightarrow[\mathcal{R}]{} \dots.$$

It has been briefly pointed out in the introduction chapter, that termination is an important property of TRS, and it is natural to ask whether for a given TRS, it is terminating or not. However this problem is undecidable in general (Theorem 2.2.10), i.e. there cannot be a general procedure such that for a given arbitrary TRS $\mathcal{R}$, it answers "yes" if $\mathcal{R}$ is terminating, and "no" if $\mathcal{R}$ is not terminating. However there exist numerous methods that are developed to show termination of TRSs. Of course the undecidability of termination shows that these methods cannot succeed for all terminating TRSs.

According to the survey paper [Zan00], roughly there are three types of methods to show termination of TRSs: semantical, syntactical, and transformational methods.

- Semantical methods first interpret the terms within some algebra equipped with a well-founded order, such that for every reduction step of $\mathcal{R}$, the corresponding interpreted terms are decreasing with respect to the defined order. Some special cases of these methods are *polynomial interpretations* and *matrix interpretations*. We will cover these methods in the next section. Here, the problem of choosing the suitable interpretations is approached by some available heuristic techniques.

- Syntactical methods usually define orders on terms inductively by looking at the structure of the terms. Many syntactical methods define this order by first giving the precedence on the function symbols, and then define the orders on terms recursively. They restrict a term to be always greater than its proper subterm. These methods try to find such an order such that every reduction step of $\mathcal{R}$ causes a decrease. These method are easy to implement, but only able to prove termination of some restricted classes of TRSs. Some special cases of these methods are *simplification orders* like *recursive path orders* (RPOs) and *Knuth-Bendix orders* (KBOs).

- Transformational methods differs from the semantical and syntactical methods. These methods prove termination of $\mathcal{R}$ by transforming it to $\Phi(\mathcal{R})$ such that the termination and non-termination of $\mathcal{R}$ can be proved by any other available methods for $\Phi(\mathcal{R})$. Some special cases of these methods are the *match-bounds*, *semantic labeling*, and the *dependency pair framework* (DP framework). We will cover the methods: match-bounds and semantic labeling in this chapter, while the DP framework is covered in the next chapter as the main part of this thesis.

This chapter will shortly survey some general termination proof methods from each of these categories. For each method we will mention its basic idea, the underlying theory, and give some examples to illustrate the applications. The purpose of this short survey is to support the next chapter of the DP framework as a method that can incorporate other termination proof methods, including some methods mentioned in this chapter. Hence we will only select a small number of the methods for this purpose. This chapter is organized as follows: Section 3.1 will briefly mentions the semantical and syntactical methods to construct reduction orders, like polynomial interpretations, matrix interpretations, and simplification orders. Section 3.2 and 3.3 will briefly survey the transformational methods: the match-bounds and the semantic labeling methods.

3.1 Reduction Orders

As indicated in Theorem 2.1.9, the basic idea to prove termination is by finding a well-founded ordering $>$ on terms such that every reduction step causes a decrease with respect to the order, i.e. if we can show for all s, t, $s \xrightarrow[\mathcal{R}]{} t$ implies $s > t$, then we can prove that $\mathcal{R}$ is terminating. However proving this property will involve infinitely many pairs s, t to be checked. We would like to check just for finitely many rewrite rules $l \to r \in \mathcal{R}$ whether $l > r$. Hence, in order to imply this property, the order $>$ must satisfy some additional properties. This motivates the definition of *reduction orders*.

Definition 3.1.1 (reduction orders)**.**
A strict order $>$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is called a *reduction order* iff

- $>$ is closed under context: $s > t$ implies $f(t_1, \ldots, s, \ldots, t_n) > f(t_1, \ldots, t, \ldots, t_n)$ for all $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and function symbol $f \in \mathcal{F}^n$, where $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{V})$.
- $>$ is closed under substitutions: $s > t$ implies $\sigma(s) > \sigma(t)$ for all $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and substitution σ.
- $>$ is a well-founded.

Theorem 3.1.2. *TRS $\mathcal{R}$ is terminating if and only if there exists a reduction order $>$ such that $l > r$ for all $l \to r \in \mathcal{R}$.*

Proof.

- ($\Rightarrow$) If $\mathcal{R}$ is terminating, then the reduction $\xrightarrow[\mathcal{R}]{+}$ itself is already a reduction order that satisfies $l \xrightarrow[\mathcal{R}]{+} r$ for all $l \to r \in \mathcal{R}$.
- ($\Leftarrow$) Suppose $>$ is the reduction order. Since $l > r$, then $t[\sigma(l)]_p > t[\sigma(r)]_p$ for all terms t, substitutions σ, and positions $p \in \mathcal{P}os(t)$. Hence, if $s \xrightarrow[\mathcal{R}]{} t$ then $s > t$. Moreover, since $>$ is well-founded, then there cannot be an infinite reduction $t_1 \xrightarrow[\mathcal{R}]{} t_2 \xrightarrow[\mathcal{R}]{} t_3 \xrightarrow[\mathcal{R}]{} \ldots$ Thus $\mathcal{R}$ is terminating.

□

Example 3.1.3.

For example, consider a TRS that consists of two rules $\mathcal{R} = \{f(x, f(y, x)) \to f(x, y), f(x, x) \to x\}$. We can show that $\mathcal{R}$ is terminating by defining a reduction order, such that the two rewrite rules are decreasing with respect to this order. This reduction order is defined as following,

$$s > t \text{ iff } \mid s \mid > \mid t \mid \text{ and for every variable } x, \mid s \mid_x \geq \mid t \mid_x$$

This order is a reduction order for terms of $\mathcal{R}$, since for every $s > t$, then $f(s, u) > f(t, u)$ and $f(u, s) > f(u, s)$ for some term u, and $s\sigma > t\sigma$ for some substitution σ. Moreover this order is also well-founded because the size of terms is well-founded. Hence we can use this order to show that the rewrite rules of $\mathcal{R}$ are strictly decreasing:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\mid f(x, f(y, x)) \mid = 5 & > & 3 = \mid f(x, y) \mid & \mid f(x, x) \mid = 3 & > & 1 = \mid x \mid \\
\mid f(x, f(y, x)) \mid_x = 2 & \geq & 1 = \mid f(x, y) \mid_x & \mid f(x, x) \mid_x = 2 & \geq & 1 = \mid x \mid_x \\
\mid f(x, f(y, x)) \mid_y = 1 & \geq & 1 = \mid f(x, y) \mid_y & \mid f(x, x) \mid_y = 0 & \geq & 0 = \mid x \mid_y
\end{array}$$

Hence we obtain $f(x, f(y, x)) > f(x, y)$ and $f(x, x) > x$. By Theorem 3.1.2 we can conclude that $\mathcal{R}$ is terminating.

Reduction orders are an important tool for proving termination of TRSs. The main problem here is to choose an appropriate reduction order that can show termination. In practical, of course the undecidability of termination shows that there is no automated method to construct reduction orders such that it will succeed for all terminating TRS. Nevertheless, it is possible to develop methods that can construct reduction orders to prove termination of some particular TRSs. In the following, we select some of the methods to construct reduction orders. We will shortly introduce their basic idea and theory, then give some examples to explain the methods. The first two methods: *polynomial interpretations* and *matrix interpretations*, can be categorized as semantical methods. They are basically trying to find an interpretation on terms to obtain the reduction orders.

3.1.1 Polynomial Interpretations

The *polynomial interpretation* is one of the methods that can construct the reduction orders to prove termination. In order to construct the reduction order, this method considers the interpretations of the terms in some algebra $\mathcal{A}$ that is equipped with some well-founded order. Then

to show some term s is decreasing to t, this method looks at the interpretation of s and t in $\mathcal{A}$. Some variables may occur in s or t. For this reason, before interpeting some term in $\mathcal{A}$, we define a variable assignment α that will assign each occuring variable to some element in the domain of $\mathcal{A}$.

Definition 3.1.4 (algebras, interpretations, well-founded monotone algebras)**.**

- An *$\mathcal{F}$-algebra* $\mathcal{A} = (A, \cdot_{\mathcal{A}})$ consists of a domain A, and a mapping $\cdot_{\mathcal{A}}$ that associates to each function symbol $f \in \mathcal{F}^n$ a mapping $f_{\mathcal{A}} : A^n \to A$.
- Let $\mathcal{A} = (A, \cdot_{\mathcal{A}})$ be an *$\mathcal{F}$-algebra*. The *interpretation function* under the variable assignment $\alpha : \mathcal{V} \to A$ is a mapping $[\,]^{\mathcal{A},\alpha} : \mathcal{T}(\mathcal{F}, \mathcal{V}) \to A$, where

$$[t]^{\mathcal{A},\alpha} = \begin{cases} \alpha(t) & \text{if } t \text{ is variable} \\ f_{\mathcal{A}}([t_1]^{\mathcal{A},\alpha}, \ldots, [t_n]^{\mathcal{A},\alpha}) & \text{if } t = f(t_1, \ldots t_n) \end{cases}$$

- A *well-founded monotone algebra* $(\mathcal{A}, >)$ consists of a nonempty algebra $\mathcal{A} = (A, \cdot_{\mathcal{A}})$ and a well-founded order $>$ on A, such that $f_{\mathcal{A}}$ is *strictly monotone*: $a > b$ implies $f_{\mathcal{A}}(a_1, \ldots, a, \ldots a_n) > f_{\mathcal{A}}(a_1, \ldots, b, \ldots a_n)$ for every $a_1, \ldots, a_n, a, b \in A$.
- The strict order induced by a well-founded monotone algebra $(\mathcal{A}, >)$ is denoted by $>_{\mathcal{A}}$, where

$$s >_{\mathcal{A}} t \text{ iff } [s]^{\mathcal{A},\alpha} > [t]^{\mathcal{A},\alpha} \text{ for all variable assignment } \alpha$$

Theorem 3.1.5. *Let $(\mathcal{A}, >)$ be a well-founded monotone algebra, and $>_{\mathcal{A}}$ the order induced by $(\mathcal{A}, >)$, then $>_{\mathcal{A}}$ is a reduction order.*

This theorem said that the strict order $>_{\mathcal{A}}$ induced by a well-founded monotone algebra $\mathcal{A}$ is indeed a reduction order. Hence to show some TRS $\mathcal{R}$ is terminating, it is sufficient to find a well-founded monotone algebra $(\mathcal{A}, >)$, such that all the rewrite rules $l \to r$ are decreasing with respect to the order induced by $\mathcal{A}$, i.e. $l >_{\mathcal{A}} r$. One of the most popular techniques to find the suitable algebra is by taking the set of natural numbers as the domain, and interpreting the function symbols with some strictly monotone polynomial functions over their arguments. This method is known as *polynomial interpretation*.

Definition 3.1.6 (polynomial interpretations)**.**

A *polynomial interpretation* $\mathcal{P}$ consists of an algebra $\mathcal{A}$ where $\mathcal{A} = (A \subseteq \mathbb{N}, \cdot_{\mathcal{A}})$ and the standard order $>$ on $\mathbb{N}$, where for every function symbol $f \in \mathcal{F}^n$, $f_{\mathcal{A}} \in \mathbb{Z}[x_1, \ldots, x_n]$. Here, $\mathbb{Z}[x_1, \ldots, x_n]$ is the set of polynomials in n indeterminates: $x_1, \ldots, x_n$, with coefficients from $\mathbb{Z}$.

Example 3.1.7.

For example, consider a TRS $\mathcal{R}$ of addition and multiplication of natural numbers that consists of four rewrite rules (written in infix notations)

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

We can show that $\mathcal{R}$ is terminating by taking a polynomial interpretation $\mathcal{P}$ with domain $\mathbb{N}$, where the interpretations of the function symbols are

$$\begin{aligned} 0_{\mathcal{P}} &= 1 \\ s_{\mathcal{P}}(x) &= x + 1 \\ +_{\mathcal{P}}(x, y) &= x + 2y \\ \times_{\mathcal{P}}(x, y) &= (x + 1)(y + 1)^2 \end{aligned}$$

By taking these polynomial interpretations the interpreted left and right -hand sides of the rewrite rules are decreasing with respect to the order $>_{\mathcal{P}}$ as described in the following. Hence $\mathcal{R}$ is terminating.

$$\begin{aligned} [x + 0]_{\mathcal{P}} = x + 2 &> x = [x]_{\mathcal{P}} \\ [x + s(y)]_{\mathcal{P}} = x + 2y + 2 &> x + 2y + 1 = [s(x + y)]_{\mathcal{P}} \\ [x \times 0]_{\mathcal{P}} = 4x + 4 &> 1 = [0]_{\mathcal{P}} \\ [x \times s(y)]_{\mathcal{P}} = xy^2 + 4xy + y^2 + 4x + 4y + 4 &> xy^2 + 2xy + y^2 + 3x + 2y + 1 = [(x \times y) + x]_{\mathcal{P}} \end{aligned}$$

There are some possible heuristic approaches to obtain polynomial interpretations. Most of them may start with fixing some domain $A \subseteq \mathbb{N}$, and then give an abstract polynomial interpretation for each function symbol. The abstract polynomial interpretations can be linear, quadratic, or with higher orders. For example, consider a TRS $\mathcal{R}$ that consist of two rewrite rules:

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

In the linear case we will give an abstract polynomial to each function symbols $+$, s, and 0 with

$$\begin{aligned} +_{\mathcal{P}}(x, y) &= ax + by + c \\ s_{\mathcal{P}}(x) &= dx + e \\ 0_{\mathcal{P}} &= f \end{aligned}$$

Here $a, \ldots, f$ are coefficients, and their values are not defined yet. The goal of the heuristic approaches is to define these values. Moreover, we have to make sure that the obtained values

can give suitable interpretations to show termination of the given TRS $\mathcal{R}$. In this case we require that

$$\begin{aligned} [x+0]_{\mathcal{P}} = ax + bf + c &> x = [x]_{\mathcal{P}} \\ [x+s(y)]_{\mathcal{P}} = ax + b\cdot(dy+e) + c &> d\cdot(ax+by+c)+e = [s(x+y)]_{\mathcal{P}} \end{aligned}$$

Now the problem of searching for the reduction orders becomes the problem of solving the abstract polynomial constraints. In this example we have to search for the value of the abstract coeffients $a, \ldots, f$, such that these constraints hold over the fixed domain A. Unfortunately, in general it is undecidable whether two terms l, r satisfy $l_{\mathcal{P}} > r_{\mathcal{P}}$ for some polynomial interpretation $\mathcal{P}$, since this is a consequence of the undecidability of Hilbert's 10th Problem [Dav73]. Hence the heuristic approaches must restrict the search of the values into some fixed range. By restricting the range of the values, the search problems become finite. However the search space can be quite large. In [FGMSTZ07] one possible approach to handle this problem is introduced. First, the so-called *diophantine constraints* from the abstract polynomial constraints are extracted. Then on some fixed range of coefficients values, the *diophantine constraints* are translated into a propositional logic formula, and a SAT solver is called to solve the search problem. We will illustrate this technique as following.

Consider the previous abstract polynomial constraints. They are of the form $l_{\mathcal{P}} > r_{\mathcal{P}}$ for some rewrite rule $l \to r$. Before extracting the diophantine constraints, first we have to transform this form into the form $l_{\mathcal{P}} - r_{\mathcal{P}} > 0$, and then group the abstract coefficients over the variables, i.e.

$$(a-1)x + (bf+c) > 0 \tag{3.1}$$
$$(a-da)x + (be+c-dc-e) > 0 \tag{3.2}$$

Now the left-hand sides are polynomials whose coefficients are polynomials over the abstract coefficients. Instead of solving these inequalities, we can use the result from [HJ98] to simplify these constraints. In that paper it is said that, we can show $\mathcal{P}ol > 0$ for some polynomial $\mathcal{P}ol$, by showing the constants of $\mathcal{P}ol$ are positive and all the coefficients of $\mathcal{P}ol$ are non-negative. In this case, instead of solving the previous constraints it is sufficient to solve the following constraints

$$\begin{aligned} bf + c &> 0 \\ be + c - dc - e &> 0 \\ a - da &\geq 0 \\ a - 1 &\geq 0 \end{aligned}$$

This is so since, on the left-hand sides of (3.1) and (3.2), the polynomials $bf+c$ and $be+c-dc-e$ are constants, whereas $a-da$ and $a-1$ the coefficients of x. This form of constraints is called *diophantine constraints*.

Recall that we have fixed some range for the abstract coefficients values $a, \ldots, f$. In [FGMSTZ07] diophantine constraints are encoded over the fixed range coefficients values into some propositional logic formula. More details about the encoding can be found in the referred paper. If the formula is satisfiable then the diophantine constraints are also satisfiable in the given range. The

SAT solver will return some possible solutions which will coincide with the suitable values for the coefficients. In this case we will obtain $a = 1$, $b = 2$, $c = 1$, $d = 1$, $e = 1$, and $f = 0$ as one of the possible solutions. If we assign these values to the abstract polynomials, then we obtain polynomial interpretations that can show the original TRS is terminating.

Beside polynomial interpretations, there are also other methods that are based on interpretations, but with a different kind of domains and interpretation functions. For example, in [Sal04] some polynomial interpretation method over the real numbers is described, in [EWZ06] some matrix interpretation method over the vector of natural numbers $\mathbb{N}^d$ is described, and in [KW08] a matrix interpretation method over $\mathbb{N}^d \cup \infty$ is described. We will not present all of them here, but in the next subsection we will shortly mention the second method, i.e. the matrix interpretation method over $\mathbb{N}^d$.

3.1.2 Matrix interpretations

The idea of this method basically is the same as the polynomial interpretations method. We try to find some well-founded monotone algebra $(\mathcal{A}, >)$, such that all the rewrite rules $l \to r$ are decreasing with respect to the order induced by $\mathcal{A}$. But instead of taking the set of natural numbers as the domain, this method uses vectors of natural numbers. These vectors are ordered by some specific well-founded order but not total. For each function symbols, we interpret them as some suitable linear mappings represented by matrices. To be more precise, the following are the formal definitions of the matrix algebras, and the order on the vectors of natural numbers.

Definition 3.1.8 (matrix algebras)**.**

- A *matrix algebra* $\mathcal{M} = (\mathbb{N}^d, \cdot_{\mathcal{M}})$ is an algebra where for every function symbol $f \in \mathcal{F}^n$,

$$f_{\mathcal{M}}(\bar{x_1}, \ldots, \bar{x_n}) = M_1\bar{x_1} + \ldots + M_n\bar{x_n} + \bar{f}$$

  $M_1, \ldots, M_n$ are matrices of $\mathbb{N}^{d\times d}$ with $(M_i)_{1,1} \geq 1$ for all $1 \leq i \leq n$, and $\bar{f}$ is a vector of $\mathbb{N}^d$.

- The strict order induced by a matrix algebra $\mathcal{M}$ is denoted by $>_{\mathcal{M}}$ and defined as

$$\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} >_{\mathcal{M}} \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} \text{ iff } x_1 > y_1, x_2 \geq y_2, \ldots, x_d \geq y_d$$

Lemma 3.1.9. *For a matrix algebra $\mathcal{M}$, $(\mathcal{M}, >_{\mathcal{M}})$ is a well-founded monotone algebra.*

This lemma expresses that $(\mathcal{M}, >_{\mathcal{M}})$ is a well-founded monotone algebra for the matrix algebra $\mathcal{M}$. By considering this lemma and the result from Theorem 3.1.5 the strict order induced by some matrix algebra $\mathcal{M}$ is also a reduction order. Hence to show some TRS $\mathcal{R}$ is terminating, we have to find a matrix algebra $\mathcal{M}$, such that all the rewrite rules $l \to r$ are decreasing with respect to $>_{\mathcal{M}}$ in the interpretation $\mathcal{M}$.

Example 3.1.10.

For example, consider the TRS consisting of one single rule $\mathcal{R} = \{h(g(s(x), y), g(z, u)) \to h(g(u, s(z)), g(s(y), x))\}$. We can show that $\mathcal{R}$ is terminating by giving some matrix algebra $\mathcal{M}$, such that the rewrite rule is decreasing with respect to the order $>_{\mathcal{M}}$ (Definition 3.1.8) in the interpretation of $\mathcal{M}$. This matrix algebra $\mathcal{M}$ is defined as follows: $\mathcal{M} = (\mathbb{N}^2, \cdot_{\mathcal{M}})$, and the function symbols are interpreted by

$$\begin{aligned} h_{\mathcal{M}}(x_0, x_1) &= \begin{pmatrix} 3 & 1 \\ 0 & 1 \end{pmatrix} \cdot x_0 + \begin{pmatrix} 1 & 3 \\ 0 & 1 \end{pmatrix} \cdot x_1 + \begin{pmatrix} 0 \\ 2 \end{pmatrix} \\ g_{\mathcal{M}}(x_0, x_1) &= \begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix} \cdot x_0 + \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} \cdot x_1 \\ s_{\mathcal{M}}(x_0) &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot x_0 + \begin{pmatrix} 0 \\ 2 \end{pmatrix} \cdot x_1 \end{aligned}$$

Hence we obtain that

$$\begin{gathered} [h(g(s(x), y), g(z, u))]^{\mathcal{M},\alpha} \\ = \\ \begin{pmatrix} 7 & 3 \\ 2 & 1 \end{pmatrix} \cdot x + \begin{pmatrix} 5 & 1 \\ 1 & 0 \end{pmatrix} \cdot y + \begin{pmatrix} 5 & 1 \\ 1 & 0 \end{pmatrix} \cdot z + \begin{pmatrix} 7 & 3 \\ 2 & 1 \end{pmatrix} \cdot u + \begin{pmatrix} 6 \\ 4 \end{pmatrix} \\ >_{\mathcal{M}} \\ \begin{pmatrix} 7 & 3 \\ 2 & 1 \end{pmatrix} \cdot x + \begin{pmatrix} 5 & 1 \\ 1 & 0 \end{pmatrix} \cdot y + \begin{pmatrix} 5 & 1 \\ 1 & 0 \end{pmatrix} \cdot z + \begin{pmatrix} 7 & 3 \\ 2 & 1 \end{pmatrix} \cdot u + \begin{pmatrix} 4 \\ 2 \end{pmatrix} \\ = \\ [h(g(u, s(z)), g(s(y), x))]^{\mathcal{M},\alpha} \end{gathered}$$

Since the only rewrite rule is strictly decreasing with respect to the order induced by the matrix algebra $\mathcal{M}$, by Theorem 3.1.2 we can conclude that $\mathcal{R}$ is terminating.

The matrix interpretation method was first proposed to show termination of string rewriting systems (SRSs) in [HW06]. But in [EWZ06] this method is extended so it can also be used to prove termination of TRSs. In practical implementations, similar technique like polynomial interpretations are used to search for some suitable matrix interpretations. First some matrix dimension and matrix coefficients range have to be fixed to make the search problem finite, yet still with a huge search space. Then the search problem is solved by transforming it into a propositional logic formula and calling some SAT solver. More details about the encoding are described in [EWZ06]. If the formula is satisfiable then some suitable value for the matrix coefficients will be returned.

The next method of constructing the reduction orders is slightly different from the polynomial and the matrix interpretations method. This method can be categorized as syntactical method, since it basically defines the order on terms by recursion over the structure of the terms. Some special property that is required is that every terms is always greater than its proper subterms. This property is called *subterm property*, and this method is called *simplification orders*.

3.1.3 Simplification orders

The polynomial and matrix interpretations methods that are mentioned before, use some heuristic approaches to obtain reduction order. Different from them, the simplification orders method can be fully automatized to obtain the orders. This method restricts the orders to have the *subterm property*, i.e. a term is greater than its proper subterms. The definition of simplification orders is basically the same as of reduction orders, except that it replaces the requirement of $>$ being well-founded by the subterm property. However, it has been proved that having the subterm property does not make the order lose well-foundedness. Hence every simplification order is also a reduction order.

Definition 3.1.11 (simplification orders)**.**
A strict order $>$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is called *simplification order* iff

- $>$ is closed under context: $s > t$ implies $f(t_1, \ldots, s, \ldots, t_n) > f(t_1, \ldots, t, \ldots, t_n)$ for all $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and function symbol $f \in \mathcal{F}^n$, where $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{V})$.
- $>$ is closed under substitution: $s > t$ implies $\sigma(s) > \sigma(t)$ for all $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and substitution σ.
- $f(t_1, \ldots, t_n) > t_i$ for every function symbol $f \in \mathcal{F}^n$, where $1 \leq i \leq n$.

Theorem 3.1.12. *Every simplification order is a reduction order.*

The simplification orders method yields many classes of orders that are really suitable for implemention. Some popular ones are the *recursive path orders* (RPOs) and the *Knuth-Bendix orders* (KBOs). The basic idea of the RPO is: it compares two terms by first comparing their root symbols, and then recursively compares the sets of the direct subterms. The set of the direct subterms can be seen as multisets (unordered), an ordered tuple, or as combination of both. These variants yields different kinds of orders, the first one yields the orders called *multiset path order* (MPO), the second one *lexicographic path orders* (LPOs), and the last one RPOs with status.

Definition 3.1.13 (recursive path orders with status)**.**
Let $\mathcal{F}$ be a signature, $>$ a well-founded weak order on $\mathcal{F}$ (usually called *precedence*), and τ a *status function* on $\mathcal{F}$, $\tau : \mathcal{F} \to \{Lex, Mul\}$. The recursive path order (RPO) with status, $>^{RPO}$ is defined as:
$s = f(s_1, \ldots, s_n) >^{RPO} t$ iff

1. $s_i = t$ or $s_i >^{RPO} t$ for some $i \in \{1, \ldots, n\}$, or
2. $t = g(t_1, \ldots, t_m)$, $s >^{RPO} t_i$ for all $i \in \{1, \ldots, n\}$, and either
   a) $f > g$, or
   b) $f = g$ and $\langle s_1, \ldots, s_n \rangle >^{RPO}_{\tau(f)} \langle t_1, \ldots, t_m \rangle$

The order $>_{\tau(f)}^{RPO}$ in the second case depends on the status of f. If $\tau(f) = Lex$, then this order becomes the lexicographic extension of RPO, and if $\tau(f) = Mul$, then this order becomes the multiset extension of RPO. Hence in the RPO with status we can define for each function symbols how their direct subterms are compared, in a lexicographic way or in a multiset way. If no status is given, e.g. $\tau(f) = Lex$ (or $\tau(f) = Mul$ resp.) for every $f \in \mathcal{F}$ then the previous definition of RPOs with status become the definition of LPOs (MPOs resp.). For example, the following is the definition of LPO but in the simplified case. We compare the set of the direct subterms lexicographically from left-to-right. One may also compare them with such permutation of the arguments.

Definition 3.1.14 (lexicographic path orders (left-to-right))**.**

Let $\mathcal{F}$ be a signature, $>$ a well-founded partial order on $\mathcal{F}$. The lexicographic path order (LPO), $>_{LPO}$ is defined as:
$s = f(s_1, \ldots, s_n) >_{LPO} t$ iff

1. $s_i = t$ or $s_i >_{LPO} t$ for some $i \in \{1, \ldots, n\}$, or
2. $t = g(t_1, \ldots, t_m)$, $s >_{LPO} t_i$ for all $i \in \{1, \ldots, n\}$, and either
   a) $f > g$, or
   b) $f = g$ and there exists i, $1 \leq i \leq n$ such that $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$, and $s_i >_{LPO} t_i$.

Theorem 3.1.15. *The recursive path order (RPO) induced by some precedence on signature $\mathcal{F}$ and status function τ is a simplification order on $\mathcal{T}(\mathcal{F}, \mathcal{V})$.*

By combining this theorem with Theorem 3.1.12 and Theorem 3.1.2, we can show that some TRS $\mathcal{R}$ is terminating, by showing that there is some precedence $>$ and a status function τ, such that for every $l \to r \in \mathcal{R}$, $l >_{RPO} r$.

Example 3.1.16.

For example, consider again the TRS of addition and multiplication $\mathcal{R}$ from Example 3.1.7, whose termination has already been shown by the polynomial interpretations. We will show that $\mathcal{R}$ is terminating by using RPO.

$$x + 0 \to x \tag{3.3}$$
$$x + s(y) \to s(x + y) \tag{3.4}$$
$$x \times 0 \to 0 \tag{3.5}$$
$$x \times s(y) \to (x \times y) + x \tag{3.6}$$

First we have to give the precedence and the status function for the function symbols. Then we have to show that these four rewrite rules are decreasing with respect to the RPO induced by the given precedence and the status function. Note that the rules (3.3) and (3.5) are already decreasing with respect to some arbitrary RPOs because of the subterm property. No precedence or status function is needed to show these rules are decreasing. Then consider the rule (3.4). To

show that this rule is decreasing, we can use the precedence $+ > s$. By this precedence we only need to show that $x + s(y) >^{RPO} x + y$. However this holds since $\langle x, s(y)\rangle >^{RPO}_{\tau(+)} \langle x, y\rangle$ for both status function $\tau(+) = Lex$ and $\tau(+) = Mul$, since $s(y) >^{RPO} y$. So by using the precedence $+ > s$ without specifying the status function, we can show that (3.4) is decreasing. We can also show the last rule: (3.6) is decreasing without specifying the status function for $\times$, since $x \times s(y) >^{RPO} x \times y$ and $x \times s(y) >^{RPO} x$. Hence by taking the precedence $\times > +$ we can show (3.6) is decreasing. To sum up, we can take the precedence $\times > + > s$ with no specification for the status functions, to induce the order $>_{RPO}$ that shows that all the rewrite rules are decreasing, and hence $\mathcal{R}$ is terminating

In practice, to obtain the precedence of the function symbols, many tools collect the constraints like $f > g$, $f, g \in \mathcal{F}$, and check whether there are some possibilities to obtain a final precedence for the function symbols. In this process there may occur many possible choices, both for the precedence and the status function, but this problem can be handled with backtracking. And since the possibilities of the precedence and the status function are finite, it is decidable whether for a given finite TRS, termination can be shown by an RPO or not. However, only limited classes of terminating TRSs can be shown by this order. We call the TRSs whose termination can be shown by simplification orders *simply terminating* TRS.

The next method is a termination proof method based on a transformational technique. This method transforms the given TRS to some infinite TRS, and uses some techniques from the field of tree automata to show the *boundedness*, that will imply termination of the original TRS. This method is called *match-bounds*.

3.2 Match-bounds

In the previous section, we have seen the methods to construct reduction orders. The reduction order methods can be applied to all TRSs in general, without any exception. Different from these methods, the match-bounds method can only be used to show termination of left-linear TRSs. It was first proposed to prove termination of string rewriting systems (SRSs) [GHW03]. But then, it is shown that the method can be generalized to prove termination of left-linear TRSs [GHWZ05]. The match-bounds method first transforms the given TRS into an *enriched* TRSs, that can simulate the reduction sequences over the original TRS. Then we prove termination of the original TRS by proving termination of the enriched TRS. The following is the formal definition of the enrichments.

Definition 3.2.1 (enrichments)**.**

Let $\mathcal{R}$ be a TRS over signature $\mathcal{F}$, and $\mathcal{S}$ a TRS over signature $\mathcal{G}$. $\mathcal{S}$ is an *enrichment* of $\mathcal{R}$ if there exists a mapping $lift$: $\mathcal{T}(\mathcal{F}, \mathcal{V}) \to \mathcal{T}(\mathcal{G}, \mathcal{V})$ and a mapping $base : \mathcal{T}(\mathcal{G}, \mathcal{V}) \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that,

- $base(lift(t)) = t$ for all terms $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and $n \in \mathbb{N}$, and
- if $t_1 \xrightarrow[\mathcal{R}]{} t_2$ and $base(u_1) = t_1$, then $u_1 \xrightarrow[\mathcal{S}]{} u_2$ with $base(u_2) = t_2$.

Lemma 3.2.2. *If the enrichment $\mathcal{S}$ of a TRS $\mathcal{R}$ is terminating, then $\mathcal{R}$ is terminating.*

In the match-bounds method, we consider enrichments that have infinitely many rewrite rules over some infinite signature. However we require the enrichments to be *locally terminating*. For some (infinite) TRS with (infinite) signature, we say it is locally terminating if the restrictions to finite signatures are always terminating.

Definition 3.2.3.
Let $\mathcal{S}$ be a TRS over the (infinite) signature $\mathcal{G}$. The *restriction* of $\mathcal{S}$ to a finite signature $\mathcal{F} \subseteq \mathcal{G}$ is defined as $\{l \to r \in \mathcal{S} \mid l, r \in \mathcal{T}(\mathcal{F}, \mathcal{V})\}$. $\mathcal{S}$ is said to be *locally terminating* if every resctriction of $\mathcal{S}$ to a finite signature $\mathcal{F} \subseteq \mathcal{G}$, is terminating.

For example, consider an infinite TRS $\mathcal{R}$ = $\{f_i(x) \to f_{i+1}(x) \mid i > 0\}$ over the infinite signature $\mathcal{F}$ = $\{f_i \mid i > 0\}$. This TRS is non terminating since we have $f_i(x) \underset{\mathcal{R}}{\longrightarrow} f_{i+1}(x) \underset{\mathcal{R}}{\longrightarrow} f_{i+2}(x) \underset{\mathcal{R}}{\longrightarrow} \ldots$ But this TRS is locally terminating since if we consider only some finite signature, then there cannot be any infinite reduction sequence. Besides local termination, we also have to make sure that the enrichments are *compact* for their ground terms. Compact for the ground terms means that every reduction sequence (possibly infinite) over the enriched TRS that started from a ground term, involves only a finite signature. It has been shown that every TRS that is locally terminating and compact for the ground terms, is terminating [GHWZ05].

Definition 3.2.4.
Let $\mathcal{S}$ be a TRS over the (infinite) signature $\mathcal{G}$. $\mathcal{S}$ is *compact* for the set ground terms $\mathcal{T}(\mathcal{G})$ if $(\underset{\mathcal{S}}{\overset{*}{\longrightarrow}})(\mathcal{T}(\mathcal{G})) \subseteq \mathcal{T}(\mathcal{F})$ for a finite signature $\mathcal{F} \subseteq \mathcal{G}$.

Lemma 3.2.5. *Let $\mathcal{S}$ be a TRS over the signature $\mathcal{G}$. If $\mathcal{S}$ is locally terminating and compact for the ground terms $\mathcal{T}(\mathcal{G})$, then $\mathcal{S}$ is terminating.*

In [GHWZ05], three kinds of enrichments for left-linear TRSs are presented: *match*, *top*, and *roof*. These enrichments are some variants of the original TRS where the function symbols are labeled by natural numbers. We call the numbers *heights*. The enrichments match, top, and roof can simulate the original reduction sequence such that for every reduction step, the minimal height of the reduced subterm is increasing. For example, the enrichment match is defined such that all the right-hand sides of the rules always have one more height than the minimal height of the left-hand sides, and the enrichment top is defined such that all the right-hand sides of the rules have one more height than the top symbols of their left-hand sides. We will present this formally in the following definition. For some TRS $\mathcal{R}$, the enrichments $top(\mathcal{R})$ and $roof(\mathcal{R})$ are guaranteed to be locally terminating; and if $\mathcal{R}$ is right-linear, then $match(\mathcal{R})$ is also guaranteed to be locally terminating.

Definition 3.2.6 (match, top, roof)**.**
Let $\mathcal{R}$ be a TRS over signature $\mathcal{F}$. The extended signature $\mathcal{F}_{\mathbb{N}}$ is $\mathcal{F}_{\mathbb{N}} = \{f_n \mid f \in \mathcal{F}$ and $n \in \mathbb{N}\}$, where for f_n we call n the height of f. Let $lift_c$, $base$, and $height$ be the functions

$lift_c$: $\mathcal{T}(\mathcal{F},\mathcal{V}) \to \mathcal{T}(\mathcal{F}_{\mathbb{N}},\mathcal{V})$ for $c \in \mathbb{N}$, $base$: $\mathcal{T}(\mathcal{F}_{\mathbb{N}},\mathcal{V}) \to \mathcal{T}(\mathcal{F},\mathcal{V})$, and $height$: $\mathcal{T}(\mathcal{F}_{\mathbb{N}},\mathcal{V})$ $\to \mathbb{N}$, where for every terms $s \in \mathcal{T}(\mathcal{F},\mathcal{V})$ and $t \in \mathcal{T}(\mathcal{F}_{\mathbb{N}},\mathcal{V})$,

$$lift_c(s) = \begin{cases} s & \text{if } s \text{ is variable} \\ f_c(lift_c(s_1),\ldots,lift_c(s_n)) & \text{if } s = f(s_1,\ldots,s_n) \end{cases}$$

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

and $height(f_c(t_1,\ldots,t_n)) = c$. The TRS $match(\mathcal{R})$, $top(\mathcal{R})$, and $roof(\mathcal{R})$ are TRSs over the infinite signature $\mathcal{F}_{\mathbb{N}}$, and they are defined as follows

- The TRS $match(\mathcal{R})$ consists of all rules $l \to lift_c(r)$ such that
  - $base(l) \to r \in \mathcal{R}$, and
  - $c = 1 + min\{height(l\ |_p) \mid p \in \mathcal{FP}os(l)\}$
- The TRS $top(\mathcal{R})$ consists of all rules $l \to lift_c(r)$ such that
  - $base(l) \to r \in \mathcal{R}$, and
  - $c = 1 + height(l\ |_\epsilon)$
- The TRS $roof(\mathcal{R})$ consists of all rules $l \to lift_c(r)$ such that
  - $base(l) \to r \in \mathcal{R}$, and
  - $c = 1 + min\{height(l\ |_p) \mid p \in \mathcal{RP}os_{\mathcal{V}ar(r)}(l)\}$, where $\mathcal{RP}os_V(t) = \{p \in \mathcal{FP}os(t) \mid V \subseteq \mathcal{V}ar(t\ |_p)\}$

Example 3.2.7.

For example, consider the TRS $\mathcal{R}$ = $\{f(g(a,y),y) \to h(f(x,y))\}$. The infinite TRS consisting of the following rules is $match(\mathcal{R})$, because all the function symbols on the right-hand sides have one more height than the minimal height of the function symbols on the left-hand sides.

$$\begin{aligned} f_0(g_0(a_0,x),y) &\to h_1(f_1(x,y)) \\ f_0(g_0(a_1,x),y) &\to h_1(f_1(x,y)) \\ f_2(g_1(a_1,x),y) &\to h_2(f_2(x,y)) \\ f_7(g_4(a_3,x),y) &\to h_4(f_4(x,y)) \\ &\vdots \end{aligned}$$

The TRS consisting of the following rules is $top(\mathcal{R})$, because all of the function symbols on the right-hand sides have one more height than the height of the top symbols on the left-hand sides.

$$\begin{aligned} f_0(g_0(x,y),a_0) &\to h_1(f_1(x,y)) \\ f_0(g_0(x,y),a_1) &\to h_1(f_1(x,y)) \\ f_2(g_1(x,y),a_1) &\to h_3(f_3(x,y)) \\ f_7(g_4(x,y),a_3) &\to h_8(f_8(x,y)) \\ &\vdots \end{aligned}$$

The TRS consisting of the following rules is $roof(\mathcal{R})$, because all of the function symbols on the right-hand sides have one more height than the minimal height of the top symbols of the subterms on the left-hand sides that contain variables x, y (in [MK09] this function symbols are called *roof symbols*).

$$\begin{aligned} f_0(g_0(x,y),a_0) &\to h_1(f_1(x,y)) \\ f_0(g_0(x,y),a_1) &\to h_1(f_1(x,y)) \\ f_2(g_1(x,y),a_1) &\to h_2(f_2(x,y)) \\ f_7(g_4(x,y),a_3) &\to h_5(f_5(x,y)) \\ &\vdots \end{aligned}$$

Lemma 3.2.8. *Let $\mathcal{R}$ be a left-linear TRS.*

- *$match(\mathcal{R})$, $top(\mathcal{R})$, and $roof(\mathcal{R})$ are enrichments of $\mathcal{R}$.*
- *$top(\mathcal{R})$ and $roof(\mathcal{R})$ are locally terminating, and if $\mathcal{R}$ is right-linear, then $match(\mathcal{R})$ is locally terminating.*

This lemma says that if some TRS is left-linear, then the enrichments top and $roof$ are locally terminating; if it is linear, then the enrichment $match$ is locally terminating. By Lemma 3.2.5 the only property that we need to show, to conclude termination of the enrichments $match$, top, and $roof$, is compactness for ground terms. There is a sufficient condition to show compactness of $match(\mathcal{R})$, $top(\mathcal{R})$, or $roof(\mathcal{R})$ for ground terms. This sufficient condition is called *e-boundedness*, $e \in \{match, top, roof\}$. To prove that the enrichment $e(\mathcal{R})$ is e-bounded, we have to find some number $c \in \mathbb{N}$, called *bound*, such that for every term that can be reached from $lift_0(\mathcal{T}(\mathcal{F})) = \{lift_0(t) \mid t \in \mathcal{T}(\mathcal{F})\}$ by the enrichment $e(\mathcal{R})$, has height at most c. If this is the case, then we call $\mathcal{R}$ *e-bounded* by c. The e-boundedness is a sufficient condition to show that some enrichment $e(\mathcal{R})$, $e \in \{match, top, roof\}$, is compact for the ground terms $\mathcal{T}(\mathcal{F}_{\mathbb{N}})$. So by Lemma 3.2.5, we can prove that some left-linear TRS $\mathcal{R}$ is terminating by proving either $\mathcal{R}$ is roof- or top- bounded; or if $\mathcal{R}$ is linear, then we prove it by match-boundedness.

Definition 3.2.9 (match-bounded, top-bounded, roof-bounded)**.**
Let $e \in \{match, top, roof\}$. The TRS $\mathcal{R}$ over a signature $\mathcal{F}$ is e-bounded if $(\xrightarrow[e(\mathcal{R})]{*})(lift_0(\mathcal{T}(\mathcal{F}))) \subseteq \mathcal{T}(\mathcal{F}_{\{0,\ldots,c\}})$ for some $c \in \mathbb{N}$.

Theorem 3.2.10. *If a left-linear TRS $\mathcal{R}$ is either top-bounded, roof-bounded, or linear and match-bounded, then $\mathcal{R}$ is terminating.*

The main problem in the practical application of this method is to show the e-boundedness, $e \in \{match, top, roof\}$. In [GHWZ05], an approach by using tree automata is given. It is shown that some TRS is e-bounded by showing that there exists a finite tree automaton that is closed under the enrichment $e(\mathcal{R})$, and accepts the ground terms from $lift_0(\mathcal{T}(\mathcal{F}))$. We will not give the complete procedure of constructing the tree automaton, but we will give some example of it. More details about the procedure are described in [GHWZ05].

Example 3.2.11.

For example, consider a TRS that consists of one rewrite rule $\mathcal{R} = \{f(f(x,a),a) \to f(x,f(x,a))\}$. This TRS is left-linear but not right-linear. Hence by Theorem 3.2.10, we can show that $\mathcal{R}$ is terminating by either using the enrichment $roof(\mathcal{R})$ or $top(\mathcal{R})$, but not $match(\mathcal{R})$. We will show that $\mathcal{R}$ is terminating by using the enrichment $roof(\mathcal{R})$. The enrichment $roof(\mathcal{R})$ will consist of rules of the form

$$\begin{aligned} f_0(f_0(x,a_0),a_0) &\to f_1(x,f_1(x,a_1)) \\ f_1(f_1(x,a_0),a_0) &\to f_2(x,f_2(x,a_2)) \\ f_1(f_4(x,a_2),a_0) &\to f_2(x,f_2(x,a_2)) \\ &\vdots \end{aligned}$$

If we start a reduction sequence from the terms in $lift_0(\mathcal{T}(\mathcal{F}))$, then all the terms that we can reach by $roof(\mathcal{R})$ will have the maximum height 1. This is so since we can find a finite tree automaton that accepts $lift_0(\mathcal{T}(\mathcal{F}))$, and every term that can be reached by $roof(\mathcal{R})$ from $lift_0(\mathcal{T}(\mathcal{F}))$, is also accepted by the tree automaton. This tree automaton $\mathcal{A}$ is defined as follows: it consists of three states 0, 1, and 2, where 0 is the only final state, and it consists of six transition rules:

$$\begin{aligned} a_0 &\to 0 \\ f_0(0,0) &\to 0 \\ a_1 &\to 1 \\ f_1(0,1) &\to 2 \\ f_1(0,2) &\to 0 \\ f_1(0,2) &\to 2 \end{aligned}$$

This automaton accepts all the ground terms $lift_0(\mathcal{T}(\mathcal{F}))$ because of the first two transition rules. This automaton also accepts every term that can be reached from $lift_0(\mathcal{T}(\mathcal{F}))$ by $roof(\mathcal{R})$. For example from $f_0(f_0(a_0,a_0),a_0) \in lift_0(\mathcal{T}(\mathcal{F}))$ we have,

$$f_0(f_0(a_0,a_0),a_0) \underset{roof(\mathcal{R})}{\longrightarrow} f_1(a_0,f_1(a_0,a_1))$$

The term $f_1(a_0, f_1(a_0, a_1))$ is accepted by the automaton, since $f_1(a_0, f_1(a_0, a_1)) \xrightarrow[\mathcal{A}]{*} 0$, and 0 is the final state. This also holds in general for every term that can be reached from $lift_0(\mathcal{T}(\mathcal{F}))$ by $roof(\mathcal{R})$. We can conclude that the left-linear TRS $\mathcal{R}$ is roof-bounded, and hence by Theorem 3.2.10: $\mathcal{R}$ is terminating.

The next method is also based on a transformational technique. In this case the given TRS is transformed by first defining some semantics for each function symbols. Then the TRS is transformed by giving some labels to the function symbols in the rewrite rules, based on the defined semantics. This transformation is done such that the original TRS is terminating if and only if the labeled TRS is terminating. Some termination proof techniques that cannot be used for the original TRS may successfully be used in the labeled TRS to show termination. This method is called *semantic labeling*.

3.3 Semantic Labeling

The semantic labeling method was introduced in the paper [Zan95]. The idea of this method is to label the rewrite rules such that some simpler termination proof methods become applicable in the transformed/labeled TRS. This approach is often successful if proving termination of the labeled TRS is easier than proving termination of the original TRS. For example, non-simply terminating TRSs are often transformed to a TRS for which termination is easily be proved by any simplification order method. In the beginning of this section we will give the basic idea and the formal definition of semantic labeling, and then we will give some examples to illustrate the practical application of the semantic labeling method to prove termination.

Formally, the semantic labeling method first tries to find an $\mathcal{F}$-algebra for a given TRS $\mathcal{R}$, to interpret terms. Then, under this interpretation, it tries to define a labeling ℓ for each function symbols. The labeling ℓ is defined by choosing for every $f \in \mathcal{F}$, a corresponding set of labels L_f and a mapping ℓ_f, which describes how the terms $f(t_1, \ldots, t_n)$ can be labeled. Then to obtain the labeled TRS $\mathcal{R}_{lab}$, we simply label the left and the right -hand sides of the rewrite rules of $\mathcal{R}$ by ℓ.

Definition 3.3.1 (labeling, labeled term, labeled TRS)**.**

- A labeling ℓ for an $\mathcal{F}$-algebra $\mathcal{A}$ consists of
  - sets of labels $L_f \subseteq A$ for every $f \in \mathcal{F}$, and
  - a labeling functions $\ell_f : A^n \to L_f$ for every n-ary $f \in \mathcal{F}$ where $L_f \neq \emptyset$.
- Let ℓ be a labeling, and α a variable assignment. The labeled term t by ℓ under α is defined recursively as follows:

$$lab_\alpha(t) = \begin{cases} t & \text{if } t \in \mathcal{V} \\ f(lab_\alpha(t_1), \ldots, lab_\alpha(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } L_f = \emptyset \\ f_{\ell_f([\alpha]_{\mathcal{A}}(t_1), \ldots, [\alpha]_{\mathcal{A}}(t_n))}(lab_\alpha(t_1), \ldots, lab_\alpha(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } L_f \neq \emptyset \end{cases}$$

- The labeled TRS $\mathcal{R}$ by ℓ is $\mathcal{R}_{lab} = \{lab_\alpha(l) \to lab_\alpha(r) \mid l \to r \in \mathcal{R}$ with $\alpha : \mathcal{V} \to A$ variable assignment $\}$

In the paper [Zan95], two variants of the semantic labeling method are introduced. The first one requires the algebra to be a *model* of $\mathcal{R}$, whereas the second one requires the algebra to be a *quasi-model* of $\mathcal{R}$. We say that an algebra $\mathcal{A}$ is a *model* of some TRS $\mathcal{R}$, if for all rules $l \to r \in \mathcal{R}$, $l =_{\mathcal{A}} r$. For a weakly monotone algebra $\mathcal{A}$ that is equipped with some partial order $\geq$, we say that $\mathcal{A}$ is a *quasi-model* of some TRS $\mathcal{R}$ if for all rules $l \to r \in \mathcal{R}$, $l \geq_{\mathcal{A}} r$. The main result from the semantic labeling method says that, if we can find an algebra $\mathcal{A}$ and a labeling ℓ such that $\mathcal{A}$ is a model for some TRS $\mathcal{R}$, then proving termination of $\mathcal{R}$ is equivalent to proving termination of the labeled TRS $\mathcal{R}_{lab}$. We also may weaken the restriction of model to a quasi-model, but there is some additional labeled TRS that has to be proved terminating. Formally these results are formulated in the following theorem.

Theorem 3.3.2 (model version, quasi-model version)**.**

- *Let $\mathcal{R}$ be a TRS, $\mathcal{A}$ a non-empty algebra model of $\mathcal{R}$, and ℓ a labeling for $\mathcal{A}$. The TRS $\mathcal{R}$ is terminating if and only if the TRS $\mathcal{R}_{lab}$ is terminating.*
- *Let $\mathcal{R}$ be a TRS, $\mathcal{A}$ a well-founded weakly monotone algebra quasi-model for $\mathcal{R}$, and ℓ a labeling for $\mathcal{A}$. The TRS $\mathcal{R}$ is terminating if and only if the TRS $\mathcal{R}_{lab} \cup \mathcal{D}ec$ is terminating.*

$$\mathcal{D}ec = \{f_a(x_1, \ldots, x_n) \to f_b(x_1, \ldots, x_n) \mid a, b \in L_f \text{ with } a > b\}. \tag{3.7}$$

The second variant of the semantic labeling techniques weakens the restriction from *model* to *quasi-model*, but requires to show that the TRS $\mathcal{R}_{lab} \cup \mathcal{D}ec$ is also terminating. The following are two examples. The first one use the sematic labeling method which restricts the algebra to be a model. The second one shows a case where restricting the algebra to be a model will not succeed, but weakening the restriction to be a quasi-model will.

Example 3.3.3.

Consider the Toyama's TRS, $\mathcal{R} = \{f(0, 1, x) \to f(x, x, x)\}$. Suppose we interpret $\mathcal{R}$ with an algebra $\mathcal{A} = \{A, \cdot_{\mathcal{A}}\}$, where the domain is $A = \{0, 1\}$, and we interpret the function symbols 0, 1, and f, with $0_{\mathcal{A}} = 0$, $1_{\mathcal{A}} = 1$, and $f_{\mathcal{A}}(x, y, z) = 0$. By a simple observation we can see that $\mathcal{A}$ is a model of $\mathcal{R}$. Now suppose we define a labeling ℓ, where the sets of labels are $L_f = \{0, 1\}$ and $L_0 = L_1 = \emptyset$, and the mapping that describes how we can label the function f, $\ell_f(x, y, z) = \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{if } x = y \end{cases}$. By the labeling ℓ, the labeled left- and right- hand sides of the rule are $lab_\alpha(f(0, 1, x)) = f_1(0, 1, x)$, and $lab_\alpha(f(x, x, x)) = f_0(x, x, x)$. We obtain the labeled TRS, $\mathcal{R}_{lab} = \{f_1(0, 1, x) \to f_0(x, x, x)\}$, which is terminating. By Theorem 3.3.2 (first part), we can conclude that the original TRS $\mathcal{R}$ is also terminating.

Example 3.3.4.

Consider a TRS $\mathcal{R}$, a variant of the previous TRS, that consists of four rewrite rules,

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

Suppose we want to use the first part of Theorem 3.3.2. Hence we need to find an algebra $\mathcal{A}$ that models the TRS $\mathcal{R}$, i.e. for every rewrite rules $l \to r$, the interpretation of the left and the right -hand sides must be equal, i.e. $l =_{\mathcal{A}} r$. But this will imply that every term must have the same interpretation as c. The function symbols f in the first rule will have the same label for the left and the right hand -sides. In this case, proving termination of the labeled TRS is not easier than proving termination of the original TRS. Hence, restricting the algebra to be a model will not work. However, weakening the restriction to be a quasi-model will ease proving termination.

Consider a weakly monotone algebra $\mathcal{A}$ with three elements, $A = \{0, 1, 2\}$ where the order $>$ on A is $2 > 0$, $1 > 0$, and consider the interpretation of the function symbols with $a_{\mathcal{A}} = 1$, $b_{\mathcal{A}} = 2$, and $c_{\mathcal{A}} = f_{\mathcal{A}}(x, y, z) = 0$. Hence $\mathcal{A}$ is a quasi-model of $\mathcal{R}$ since for every $l \to r \in \mathcal{R}$, $l \geq_{\mathcal{A}} r$. Now suppose we define a labeling ℓ, where the set of labels are $L_f = \{0, 1\}$ and $L_a = L_b = L_c = \emptyset$, and the mapping that describes how we can label the function f, $\ell_f(x, y, z) = \begin{cases} 1 & \text{if } x = 1 \text{ and } y = 2 \\ 0 & \text{if otherwise} \end{cases}$. By this labeling, the labeled TRS $\mathcal{R}_{lab}$ consists of

$$\begin{aligned} f_1(a, b, x) &\to f_0(x, x, x) \\ f_0(x, y, z) &\to c \\ f_1(x, y, z) &\to c \\ a &\to c \\ b &\to c \end{aligned}$$

and the TRS $\mathcal{D}ec$ will consist of only one rule: $f_1(x, y, z) \to f_0(x, y, z)$. The combined TRS, $\mathcal{R}_{lab} \cup \mathcal{D}ec$ can be easily shown terminating. By the second part of Theorem 3.3.2 we can conclude that the original TRS $\mathcal{R}$ is also terminating.

In order to implement the semantic labeling method, we have to search for some interpretations and the labeling functions such that we can prove that the labeled TRS is terminating. In [KZ06], one possible approach to do this search is shown. First the interpretation domain is restricted to the boolean value or natural numbers, and the interpretation of every function symbols is restricted to a polynomial or a min/max function. Furthermore, the labeling function ℓ_f for every function symbol f, is also restricted to the identity function. Then such interpretations and labeling functions are searched, such that all of the labeled rewrite rules are decreasing with respect to some RPO. The main result of [KZ06] is the procedure how to do this. In that paper, it is shown that the semantic labeling technique still can be automated in combination with RPO, even though the model consists of an infinite set of natural numbers and infinite signatures. Despite using such strict restrictions for the interpretations and the labeling functions, the author shows some successful examples in [KZ06], where their techniques can easily yield termination proofs of some TRSs but in which all other known methods fail.

In this chapter we have already shown briefly some general termination proof methods. We have shown the methods based on semantical or syntactical approaches like the reduction orders (polynomial interpretations, matrix interpretations, simplification orders), and the methods with transformational approach: match-bounds and semantic labeling. We will show in the next chapter that the methods mentioned here can also be incorporated in some general termination

proof method called the *dependency pair framework* (DP framework). The DP framework can be categorized as a transformational method, since first it transforms the given TRS into a form of a pair of TRSs, and then tries to prove termination on these pairs. We will describe more details about this method in the next chapter.

CHAPTER 4

The Dependency Pair Framework

There are many possible methods to show termination of TRSs. As we can see from the previous chapter, there are semantical and syntactical methods like reduction orders (polynomial interpretations, matrix interpretations, simplification orders), and there are also some modern approaches based on transformations like match-bounds and semantic labeling. Despite the existence of various termination proof methods, there are still many important TRSs that are difficult or cannot be shown terminating by some of these methods directly. This problem motivates the development of a new and powerful method called the *dependency pair framework* (DP framework). In this case, we can classify the DP framework as a transformational method, since basically it does not prove termination directly on some TRS but first transforms the TRS into a pair of TRSs, and then tries to prove termination on these pairs.

The basic idea of the DP framework was introduced in [AG00], and generally formulated in [GTS05], where the authors use the notion of the *dependency pairs* as an approach to prove termination. The dependency pairs basically are pairs that compare the left-hand sides of the rules with some special subterms of their right-hand sides. Based on the notion of the dependency pairs, a new termination criterion that is really suitable for automation is introduced. The termination criterion basically says that a given TRS is terminating if and only if it does not admit infinite sequences of dependency pairs, whereas in the middle of their applications we still allow some finite applications of the rewrite rules. This termination criterion is the theoretical foundation of the DP framework, that leads us to consider two different TRSs instead of a single one. The framework constructs a pair of TRSs that consist of the original TRS and its dependency pairs at the beginning. On these initial pairs the DP framework will start the termination proofs by simplifying them gradually.

The main idea of the framework is simple, it tries to simplify the termination (finiteness) proof of the initial pairs repeatedly. We start from the initial pairs, and then to prove termination (finiteness) of some pair, we transform it into other (hopefully simpler) pairs. We will stop if all the pairs derived from the initial one can be shown terminating (finite). This transformations is done by the so-called *dependency pair processors* (DP processors). In each steps of transformations, different DP processors can be applied. Each of the DP processors may also incorporate

different techniques from the general termination proof methods such as polynomial interpretations, matrix interpretations, RPOs, KBOs, semantic labeling, match-bound, etc. This idea increases significantly the flexibility, modularity, and the power of proving termination. Many TRSs that are difficult or cannot be shown terminating with the direct methods like reduction orders and simplification orders now can be shown easily by this framework.

The main and the most important component of the DP framework are the DP processors, since basically the DP framework is nothing more but a repeated application of the DP processors. We will include in this chapter some comprehensive summary of important types of DP processors. This chapter is organized as follows: In Section 4.1 we will give the basic idea of the dependency pairs and some theoretical background why they can be used as an approach to prove termination. Then in Section 4.2 we will describe the basic idea of the DP framework and its formulation. In this part we will mention formally, what the DP framework, DP problems, and the DP processors are. In Section 4.3 we will give a comprehensive summary of some various important types of DP processors. We work out the basic underlying ideas and theory of each processors, point out their successfulness, present some refinements, and illustrate the application by examples. We will also illustrate some running example of proving termination with the DP framework along the summary of the DP processors.

4.1 Dependency Pairs

As it has been mentioned shortly before, by using the definition of the *dependency pairs* we can prove termination of TRSs by proving that there cannot be infinite applications of the dependency pairs, whereas in the middle of their applications we still allow some finite applications of the rewrite rules. We will briefly give the basic idea of the dependency pairs and why this is the case.

Suppose a TRS $\mathcal{R}$ is non-terminating, then there exists an infinite reduction chain,

$$t_1 \xrightarrow[\mathcal{R}]{} t_2 \xrightarrow[\mathcal{R}]{} t_3 \xrightarrow[\mathcal{R}]{} \dots.$$

By a minimality argument, the term t_1 must contain a subterm $f(u_1, \dots, u_n)$, that starts an infinite reduction chain, but none of the $u_1, \dots, u_n$ has this property. We call the term $f(u_1, \dots, u_n)$ a *minimal non-terminating term*. Hence for every non-terminating TRS, there must exists a minimal non-terminating subterm.

Definition 4.1.1 (minimal non-terminating term)**.**

Let $\mathcal{R}$ be a TRS. A term t is called *minimal non-terminating* term (with respect to $\mathcal{R}$) if t is non-terminating and every proper subterm t' of t is terminating. We denote T_∞ as the set of all minimal non-terminating terms.

Lemma 4.1.2. *If a TRS $\mathcal{R}$ is non-terminating, then there exists a minimal non-terminating term.*

Now observe that for a minimal non-terminating term t there must be a rewrite rule: $l \to r \in \mathcal{R}$, that will be applied at the root position after some reduction steps below the root position of t. This is so, since $t_1, \dots, t_n$ the proper subterms of t, cannot start an infinite reduction. Hence,

there must be a reduction at the root position that starts an infinite reduction. However, before this root reduction, the proper subterms $t_1, \ldots, t_n$ may be rewritten in zero or more steps. This reductions are done below the root position of t. Suppose from this sequence of reduction steps (below and at the root position of t), we obtain a term $r\sigma$, for some substitution σ. Then $r\sigma$ must be again a non-terminating term, and there must be a non-variable u subterm of r, such that $u\sigma$ is again a minimal non-terminating term. Otherwise it contradicts that t is a minimal non-terminating term.

Lemma 4.1.3. *Let $\mathcal{R}$ be a TRS and $t \in T_\infty$, then there exist a rewrite rule $l \to r \in \mathcal{R}$, a substitution σ, and a non-variable subterm u of r such that*

$$t \xrightarrow[\mathcal{R}]{>\epsilon^*} l\sigma \xrightarrow[\mathcal{R}]{\epsilon} r\sigma \trianglerighteq u\sigma$$

where $u\sigma \in T_\infty$.

Another important point that we can derive from this observation is that, if t is a minimal non-terminating term of $\mathcal{R}$, then there must be a rewrite rule $l \to r \in \mathcal{R}$ such that the root symbol of t coincide with the root symbol of l. By defining the notion of *defined symbol*, we can formally formulate this observation.

Definition 4.1.4 (defined symbols)**.**
Let $\mathcal{R}$ be a TRS over signature $\mathcal{F}$. The function symbol $f \in \mathcal{F}$ is called *defined symbol* of $\mathcal{R}$ if $f = root(l)$, for some $l \to r \in \mathcal{R}$.

Corollary 4.1.5. *Let $\mathcal{R}$ be a TRS and T_∞ the set of all minimal non-terminating terms of $\mathcal{R}$, then for every $t \in T_\infty$, $root(t)$ is a defined symbol of $\mathcal{R}$.*

Now consider again the reduction steps that are described in Lemma 4.1.3. By Corollary 4.1.5 we know that every minimal non-terminating terms must have a defined root symbol. Then instead of having $t \xrightarrow[\mathcal{R}]{>\epsilon^*} l\sigma \xrightarrow[\mathcal{R}]{\epsilon} r\sigma \trianglerighteq u\sigma$ for some rewrite rule $l \to r$ and minimal non-terminating terms t, u, we can have $t \xrightarrow[\mathcal{R}]{>\epsilon^*} l\sigma \xrightarrow[\mathcal{S}]{\epsilon} u\sigma$ by considering some new TRS $\mathcal{S}$, which consists of new rules $l \to u$ extracted from some rewrite rule $l \to r \in \mathcal{R}$, where u is a subterm of r with a defined root symbol.

The set of the dependency pairs $\mathcal{DP}(\mathcal{R})$ for a TRS $\mathcal{R}$ is basically the same as the TRS $\mathcal{S}$. But it uses a signature extended by some new (marked) symbols for every defined symbol of $\mathcal{R}$. Hence the root symbols of the left and the right -hand sides of the dependency pairs are different from the functions symbols of the original TRS $\mathcal{R}$, and the function symbols of their proper subterm.

Definition 4.1.6 (dependency pairs)**.**
Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$,

- The extended signature $\mathcal{F}^\sharp$ is defined as $\mathcal{F}^\sharp = \mathcal{F} \cup \{f^\sharp/_n \mid f \in \mathcal{F}^n$ *defined symbol* of $\mathcal{R}\}$.

- The set of the *dependency pairs* for a TRS $\mathcal{R}$ is denoted by $\mathcal{DP}(\mathcal{R})$, and it is a TRS over the signature $\mathcal{F}^\sharp$, where $\mathcal{DP}(\mathcal{R}) = \{l^\sharp \to u^\sharp \mid l \to r \in \mathcal{R}, u \text{ subterm of } r, root(u) \textit{ defined symbol of } \mathcal{R}\}$.

In this definition, $t^\sharp$ denotes $f^\sharp(t_1, \ldots, t_n)$, for some term $t = f(t_1, \ldots, t_n)$ and some defined symbol f. To ease the readability, we may also use the capital letter F instead of $f^\sharp$.

Example 4.1.7.

For example, consider the TRS $\mathcal{R}$ of division and subtraction of natural numbers that consist of four rewrite rules

$$\begin{aligned} minus(x, 0) &\to x && (4.1)\\ minus(s(x), s(y)) &\to minus(x, y) && (4.2)\\ div(0, s(y)) &\to 0 && (4.3)\\ div(s(x), s(y)) &\to s(div(minus(x, y), s(y))) && (4.4) \end{aligned}$$

The set of the dependency pairs $\mathcal{DP}(\mathcal{R})$ consists of

$$\begin{aligned} MINUS(s(x), s(y)) &\to MINUS(x, y) && (4.5)\\ DIV(s(x), s(y)) &\to MINUS(x, y) && (4.6)\\ DIV(s(x), s(y)) &\to DIV(minus(x, y), s(y)) && (4.7) \end{aligned}$$

Now consider again the previous observation that said for some TRS $\mathcal{R}$ we can define a TRS $\mathcal{S}$, such that for every minimal non-terminating t of $\mathcal{R}$, there is a rule $l \to u \in \mathcal{S}$ such that $t \xrightarrow[\mathcal{R}]{>\epsilon^*} l\sigma \xrightarrow[\mathcal{S}]{\epsilon} u\sigma$. Here $u\sigma$ is a minimal non-terminating term. The term $l\sigma$ is also a minimal non-terminating term, since it is obtained from the minimal non-terminating term t, by reducing its proper subterm. Now by using the definition of the dependency pairs $\mathcal{DP}(\mathcal{R})$, we can replace the TRS $\mathcal{S}$, and use terms with the marked root symbols. By using the definition of the dependency pairs, we do not need to state explicitly the positions where the reductions take place, since by marking the defined symbols, the reductions $\xrightarrow[\mathcal{R}]{*}$ are always done below the root position, because $\mathcal{R}$ does not contain any rule with a marked root symbol. On the other hand, the reductions $\xrightarrow[\mathcal{DP}(\mathcal{R})]{}$ are always done at the root position, because $\mathcal{DP}(\mathcal{R})$ contains only rules with marked root symbols. We can formulate this observation in the following lemma.

Lemma 4.1.8. *Let $\mathcal{R}$ be a TRS and $\mathcal{DP}(\mathcal{R})$ the set of dependency pairs of $\mathcal{R}$. Then for every $s \in T_\infty$, there exist minimal non-terminating terms t, $u \in T_\infty$ such that*

$$s^\sharp \xrightarrow[\mathcal{R}]{*} t^\sharp \xrightarrow[\mathcal{DP}(\mathcal{R})]{} u^\sharp \qquad (4.8)$$

This result can be combined with Lemma 4.1.2, which say that every non-terminating TRS must have a minimal non-terminating term. This means that if some TRS $\mathcal{R}$ is non-terminating, then the existence of a minimal non-terminating term t, implies that there exists an infinite sequence of $\mathcal{R}$- and $\mathcal{DP}(\mathcal{R})$- steps of the form (4.8) starting from $t^\sharp$. This observation is formally given in the following Theorem.

Theorem 4.1.9 ([AG00]). *Let $\mathcal{R}$ be a TRS and $\mathcal{DP}(\mathcal{R})$ a set of dependency pairs of $\mathcal{R}$. If $\mathcal{R}$ is non-terminating then there exists a minimal non-terminating term t of $\mathcal{R}$, such that the term $t^\sharp$ starts an infinite sequence of the form*

$$t^\sharp = t_1^\sharp \xrightarrow[\mathcal{R}]{*} t_2^\sharp \xrightarrow[\mathcal{DP}(\mathcal{R})]{} t_3^\sharp \xrightarrow[\mathcal{R}]{*} t_4^\sharp \xrightarrow[\mathcal{DP}(\mathcal{R})]{} \dots \tag{4.9}$$

By using this result, we can show that $\mathcal{R}$ is terminating by showing the absence of the infinite chain (4.9). In showing this, we are interested to show that there is no infinite application of dependency pairs $\mathcal{DP}(\mathcal{R})$, whereas in the middle of their applications we still allow some finite applications of the original TRS $\mathcal{R}$. It turns out that this termination criterion is really suitable for automation. The strong and powerful termination proof method called *the dependency pair framework* (DP framework), based its termination proof on this result. More details about the DP framework will be described in the next section.

4.2 The Dependency Pair Framework

The *dependency pair framework* (DP framework) relies on Theorem 4.1.9. For a given TRS, it tries to prove the absence of infinite chains of the form (4.9). But note that in this infinite chain, we consider two different TRSs. The DP framework does not prove termination of the given TRS directly. Instead, it first transforms the TRS into a pair of TRSs, called the *dependency pair problems* (DP problems).

Definition 4.2.1 (DP problems, finite DP problems).
A *dependency pair problem* (DP problem) is a pair of TRSs $(\mathcal{P}, \mathcal{R})$. It is called *finite* if there is no infinite sequence

$$s_1 \xrightarrow[\mathcal{R}]{*} t_1 \xrightarrow[\mathcal{P}]{\epsilon} s_2 \xrightarrow[\mathcal{R}]{*} t_2 \xrightarrow[\mathcal{P}]{\epsilon} \dots \tag{4.10}$$

such that $s_1, t_1, s_2, t_2, \dots$ are terminating with respect to $\mathcal{R}$. We call this chain a *dependency pairs chain* (DP chain).

In this definition, the notion of finiteness generalizes the case of the absence of infinite chains of the form (4.9). This notion can be applied to any arbitrary pair of TRS $\mathcal{P}$, $\mathcal{R}$. Now proving termination of some TRS $\mathcal{R}$ is equivalent to showing that the initial DP problem $(\mathcal{DP}(\mathcal{R}), \mathcal{R})$ is finite.

Theorem 4.2.2 ([GTS05]). *TRS $\mathcal{R}$ is terminating if and only if the DP problem $(\mathcal{DP}(\mathcal{R}), \mathcal{R})$ is finite.*

In the DP framework, we show some DP problems are finite (initially $(\mathcal{DP}(\mathcal{R}), \mathcal{R})$) by transforming them into some other (hopefully simpler) DP problems. The transformation of the DP problems is done by the *dependency pair processors* (DP processors).

Definition 4.2.3 (DP processors).
A *dependency pair processor* (DP processor) is a function ϕ that takes a DP problem as an input and returns a new set of some other DP problems.

- The DP processor ϕ is called *sound*, if a DP problem $(\mathcal{P}, \mathcal{R})$ is finite whenever all the DP problems in $\phi((\mathcal{P}, \mathcal{R}))$ are finite.
- The DP processor ϕ is called *complete*, if a DP problem $(\mathcal{P}, \mathcal{R})$ is not finite whenever at least one of the DP problems in $\phi((\mathcal{P}, \mathcal{R}))$ is not finite.

The formulation of the DP framework is as follows. To prove a TRS $\mathcal{R}$ is terminating, we first compute the set of the dependency pairs $\mathcal{DP}(\mathcal{R})$. Then we have to show that the initial DP problem $(\mathcal{DP}(\mathcal{R}), \mathcal{R})$ is finite. In order to prove a DP problem $(\mathcal{P}, \mathcal{R})$ is finite, we apply a sound DP processor to transform it into other DP problems $(\mathcal{P}_1, \mathcal{R}_1), \ldots, (\mathcal{P}_n, \mathcal{R}_n)$. Then we show that each of them is also finite by again applying other DP processors. We can illustrate the DP framework by using a tree. The nodes are the DP problems, the root is the initial DP problem, and the children of some nodes are the DP problems derived from some transformations with sound DP processors. If we can show that all of the DP problems on the leaves are finite, then we have shown that the initial DP problem is also finite, because the requirement of soundness guaranteed that the transformations will preserve some infinite DP chain if there is any. Hence by proving that all the DP problems on the leaves are finite, we can prove the initial DP problem is also finite, and hence by Theorem 4.2.2 we have proved termination of the original TRS $\mathcal{R}$.

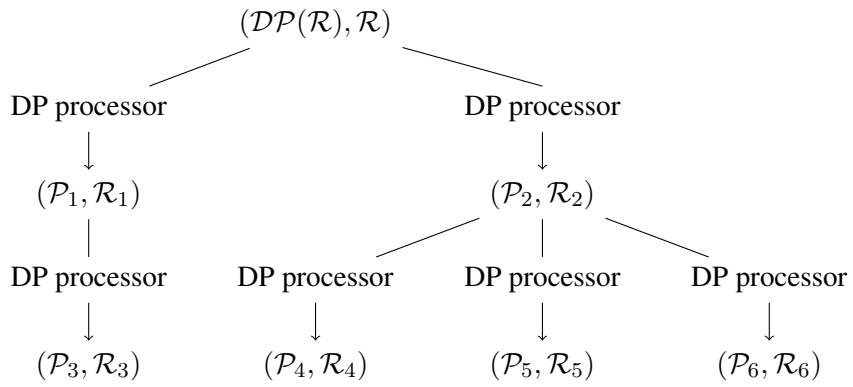

Figure 4.1: The dependency pair framework

Example 4.2.4.

For example consider Figure 4.1. Suppose we want to prove that $\mathcal{R}$ is terminating by proving finiteness of the initial DP problem $(\mathcal{DP}(\mathcal{R}), \mathcal{R})$. To prove finiteness of the $(\mathcal{DP}(\mathcal{R}), \mathcal{R})$, we have to prove finiteness of $(\mathcal{P}_1, \mathcal{R}_1)$ and $(\mathcal{P}_2, \mathcal{R}_2)$. To prove finiteness of $(\mathcal{P}_1, \mathcal{R}_1)$ we have to prove finiteness $(\mathcal{P}_3, \mathcal{R}_3)$, and to prove finiteness of $(\mathcal{P}_2, \mathcal{R}_2)$ we have to prove finiteness of $(\mathcal{P}_4, \mathcal{R}_4)$ - $(\mathcal{P}_6, \mathcal{R}_6)$. If all the DP problems on the leaves: $(\mathcal{P}_3, \mathcal{R}_3)$ - $(\mathcal{P}_6, \mathcal{R}_6)$ are finite, and all the DP processors that we used are sound, then we can conclude that the initial DP problem $(\mathcal{DP}(\mathcal{R}), \mathcal{R})$, is finite. If this is the case then we have shown that the original TRS $\mathcal{R}$ is terminating.

A similar case also applies if we want to prove non-termination. If one of the DP problem on the leaves: $(\mathcal{P}_3, \mathcal{R}_3)$ or ... or $(\mathcal{P}_6, \mathcal{R}_6)$ is not finite and all the DP processors that we use are

complete, then we can conclude that the original DP problem is not finite, and by Theorem 4.2.2 we have shown that the original TRS $\mathcal{R}$ is non-terminating. However in this summary we will only be concerned with proving termination instead of non-termination.

The DP processors that are used in the DP framework may incorporate various techniques of termination proofs. For example, we can use some techniques that are already mentioned in Chapter 2, such as polynomial interpretations, matrix interpretations, simplification orders, match-bounds, and semantic labeling. Moreover, in each transformation step we can use different DP processors and solve each DP problems separately using various techniques of termination. This idea increases significantly the flexibility, modularity, and the power of proving termination with the DP framework. Many TRSs that are difficult or cannot be shown terminating with some direct methods now can be shown easily by this framework. For example, the DP problem described in Example 4.1.7 cannot be shown terminating by using simplification orders directly. However by using the DP framework, it can be shown terminating easily. We will use this as one of the running examples in the next section.

In the DP framework, the success of proving termination cannot be separated from the various types of the DP processors. In fact, the DP processors are the most important components in the DP framework. Since basically the DP framework is nothing more but a repeated application of the DP processors. In the following section we will give a comprehensive survey of various important types of DP processors. Along the summary, we will give some examples to illustrate the use of the DP processors in termination proofs of the DP framework.

4.3 Dependency Pair Processors

As it has been introduced in Definition 4.2.3, basically a DP processor is a function ϕ that will take a DP problem $(\mathcal{P}, \mathcal{R})$ and return some DP problems $(\mathcal{P}_1, \mathcal{R}_1), \ldots, (\mathcal{P}_n, \mathcal{R}_n)$ that have to be solved instead. There are various types of DP processors. But mostly, the returned DP problems are simpler. They may contain less rewrite rules, less dependency pairs, or possibly more rules but the finiteness problems become easier to solve. All of the DP processors that we will present here are sound and complete. Hence, their applications in the DP framework can be used to prove termination (and non-termination) correctly. Some of the DP processors may use techniques from the general termination proof methods that we already described in Chapter 2. For each DP processor we work out the basic underlying ideas and theory, present some refinements, and illustrate their applications with some examples.

4.3.1 Dependency Graph Processor

This DP processor is usually applied in the first step of the DP framework. It decomposes a DP problem $(\mathcal{P}, \mathcal{R})$ into some simpler DP problems $(\mathcal{S}_1, \mathcal{R}), \ldots, (\mathcal{S}_n, \mathcal{R})$. The application of this processor allows us to prove finiteness of $(\mathcal{P}, \mathcal{R})$, by proving separately the finiteness of $(\mathcal{S}_1, \mathcal{R}), \ldots, (\mathcal{S}_n, \mathcal{R})$. The decomposition of the DP problem is based on the so-called *dependency graph*. A dependency graph is a graph where all the dependency pairs are considered as the nodes, and we will have an arrow from $s \to t$ to $u \to v$ if there exist substitutions σ, τ, such that $t\sigma \xrightarrow[\mathcal{R}]{*} u\tau$.

Definition 4.3.1 (dependency graph)**.**

Let $(\mathcal{P}, \mathcal{R})$ be a DP problem. The nodes in the *dependency graph* of the DP problem $(\mathcal{P}, \mathcal{R})$ are the dependency pairs from $\mathcal{P}$, and there is an arrow from $s \to t$ to $u \to v$ if and only if there exist substitutions σ, τ such that $t\sigma \xrightarrow[\mathcal{R}]{*} u\tau$.

Example 4.3.2.

For example, consider the DP problem from Example 4.1.7. The set of the initial dependency pairs consists of these three dependency pairs,

$$\begin{aligned} (4.5)\ MINUS(s(x), s(y)) &\to MINUS(x, y) \\ (4.6)\ DIV(s(x), s(y)) &\to MINUS(x, y) \\ (4.7)\ DIV(s(x), s(y)) &\to DIV(minus(x, y), s(y)) \end{aligned}$$

The dependency graph for this problem is depicted in Figure 4.2.

Figure 4.2: Dependency graph

Note that for some DP problem $(\mathcal{P}, \mathcal{R})$, if the application of the dependency pair $s \to t$ is followed by an application of the dependency pair $u \to v$ then, we will have an arrow from $s \to t$ to $u \to v$ in the dependency graph. This implies that, every DP chain will correspond to some path in the dependency graph. However the converse does not hold, i.e. a path in the dependency graph does not necessarily correspond to a DP chain. Moreover, since the graph is finite, every infinite DP chains must correspond to some infinite path and must involve one or more cycles in the dependency graph. Hence we can prove the absence of the infinite DP chain by disregarding the nodes that are not a part of the cycles. If the graph consists of several cycles, it is sufficient to analyze each cycle separately. However the number of cycles can be very large, even if the number of the dependency pairs is small. In the worst case, there can be $2^n - 1$ cycles for some n dependency pairs. Hence instead of considering the cycles, we consider the *strongly connected components* (SCCs). An SCC intuitively can be seen as a maximal cycle with respect to the inclusion relation. The number of SCCs is at most n, for n dependency pairs, since every dependency pair must belong to at most one SCC.

Definition 4.3.3 (strongly connected components)**.**

A graph $\mathcal{S}$ is a *strongly connected component* (SCC) in the graph $\mathcal{G}$, if $\mathcal{S}$ is a maximal subgraph of $\mathcal{G}$, such that each node in $\mathcal{S}$ has a path to every other node in $\mathcal{S}$.

Definition 4.3.4 (dependency graph processor)**.**

The *dependency graph processor* is defined by $\phi_{DG}((\mathcal{P}, \mathcal{R})) = \{(\mathcal{S}, \mathcal{R}) \mid \mathcal{S}$ is *strongly connected component* in the *dependency graph* of $(\mathcal{P}, \mathcal{R})\}$

For example, consider again the previous example of the dependency graph depicted in Figure 4.2. In this case, the SCCs are simple cycles, $\mathcal{S}_1 = \{(4.5)\}$, and $\mathcal{S}_2 = \{(4.7)\}$. Hence, to prove that the initial DP problem $(\{(4.5), (4.6), (4.7)\}), \mathcal{R})$ is finite, we can show it by proving that the simpler DP problems $(\{(4.5)\}), \mathcal{R})$ and $(\{(4.7)\}), \mathcal{R})$ are finite.

As we have mentioned before, to prove finiteness of some DP problem $(\mathcal{P}, \mathcal{R})$, it is sufficient by proving finiteness of $(\mathcal{S}_1, \mathcal{R}), \ldots, (\mathcal{S}_n, \mathcal{R})$ separately. Since if $(\mathcal{P}, \mathcal{R})$ admits an infinite DP chain, then one of $(\mathcal{S}_1, \mathcal{R}), \ldots, (\mathcal{S}_n, \mathcal{R})$ will also admit an infinite DP chain, since an infinite DP chain always corresponds to some cycle that will be contained in some SCC. Hence by Definition 4.2.3, the dependency graph processor is sound, i.e. its application preserves infinite DP chain. Moreover, if $(\mathcal{P}, \mathcal{R})$ does not admit any infinite DP chain, then none of $(\mathcal{S}_1, \mathcal{R}), \ldots, (\mathcal{S}_n, \mathcal{R})$ will admit a chain as well. Hence, we can conclude that the dependency graph processor is sound and complete.

Theorem 4.3.5. *The dependency graph processor is sound and complete.*

The main problem for the practical implementation of the dependency graph processor is that the dependency graph itself is not computable in general. Since for two arbitrary terms, t, u, it is undecidable whether there exist substitutions σ, τ, such that $t\sigma \xrightarrow[\mathcal{R}]{*} u\tau$. In practice we use an *estimated dependency graph*, that will consist of at least all of the edges from the real dependency graph. Some trivial way to estimate whether there exists an arrow from $s \to t$ to $u \to v$ is by checking whether the root symbol of t coincides with the root symbol of u. Of course this estimation may allow many arrows that do not exist in the real dependency graph. We will present two other better estimations: an estimation by unification [GTSF06] and an estimation by tree automata [Mid01].

In the estimation by unification, we will use the functions CAP and REN. For a DP problem $(\mathcal{DP}(\mathcal{R}), \mathcal{R})$, and a term t, the function $CAP(t)$ will replace the subterms of t, whose root symbols are defined symbols of $\mathcal{R}$, with distinct fresh variables. For example, consider the DP problem from Example 4.1.7, and $t = DIV(minus(x, y), s(y))$. Since $minus(x, y)$ is the subterm of t with defined root symbol,

$$CAP(DIV(minus(x, y), s(y))) = DIV(z, s(y))$$

The function $REN(t)$ replaces variables of t by distinct fresh variables (*linearization*), e.g.

$$REN(DIV(x, x)) = DIV(x_1, x_2)$$

By using these two functions REN and CAP, the estimation by unification means that there is an arrow from $s \to t$ to $u \to v$ in the estimated dependency graph iff $REN(CAP(t))$ unifies with u. We will give an example about this later, together with another estimation and their comparison.

Definition 4.3.6 (estimated dependency graph by unification [GTSF06])**.**

For a DP problem $(\mathcal{P}, \mathcal{R})$, the nodes of the *estimated dependency graph by unification* are the dependency pairs of $\mathcal{P}$, and there is an arrow from $s \to t$ to $u \to v$ if and only if $REN(CAP(t))$ and u are unifiable.

Differently from the estimation by unification which is easy to implement, the estimation by tree automata uses some procedure from the field of *tree automata* that is computationally expensive. The estimation by tree automata uses a procedure that decides whether a set of ground instances of some term intersects with some *regular tree language* or not. Recalling from the preliminaries chapter, a *regular tree language* is a set of ground terms $L \subseteq \mathcal{T}(\mathcal{F})$, such that we can find a tree automaton $\mathcal{A}$, whose accepted language is L.

In the paper [Mid01], the author tries to estimate whether there exists an arrow from $s \to t$ to $u \to v$ in the dependency graph, by combining the existing procedure to decide whether there is some intersection of ground instances of a term with some regular tree language, with the so-called *approximation mappings*. In the following we give two kinds of approximation mappings from [Mid01], that are called approximation mapping S and NV.

Definition 4.3.7 (approximation mappings S and NV)**.**

The *approximation mappings* S and NV for a TRS $\mathcal{R}$ is defined by,

- $\mathcal{R}_S = \{REN(l) \to x \mid l \to r \in \mathcal{R}, \text{and } x \text{ a fresh variable } \}$
- $\mathcal{R}_{NV} = \{REN(l) \to REN(r) \mid l \to r \in \mathcal{R}\}$

In this definition, the REN function is the *linearization* function as defined before. The approximation mapping can be seen as a mapping that throws out certain parts of terms in the rewrite rules. Throwing out certain parts is done such that the set of terms that rewrite to some regular tree language with this modified rules, is again regular, i.e. $(\xrightarrow[\mathcal{R}_\alpha]{*})[L]$ is regular, for some regular language L and approximation mapping α. The approximation mappings are defined such that we can use the procedure that decides whether there is an intersection of ground instances of a term with some regular tree language, to approximate whether there is an arrow from $s \to t$ to $u \to v$ in the dependency graph.

Definition 4.3.8 (estimated dependency graph by tree automata [[**?**]]**.**]

For a DP problem $(\mathcal{P}, \mathcal{R})$, the nodes of the *estimated dependency graph by tree automata* are the dependency pairs of $\mathcal{P}$, and there is an arc from $s \to t$ to $u \to v$ in the approximated dependency graph, if both

- $Ground(t) \cap (\xrightarrow[\mathcal{R}_S]{*})[Ground(REN(u))] \neq \emptyset$.
- $Ground(u) \cap (\xrightarrow[(\mathcal{R}^{-1})_S]{*})[Ground(REN(t))] \neq \emptyset$.

Example 4.3.9.

We will show that the estimation by the tree automata techniques (Definition 4.3.7) gives a better approximation than the estimation by unification (Definition 4.3.6). For example, consider a TRS $\mathcal{R}$ that consists of two rewrite rules:

$$\begin{array}{rcl} f(g(a)) & \to & f(a) \\ a & \to & b \end{array}$$

From these rules we obtain two dependency pairs of $\mathcal{R}$,

$$F(g(a)) \rightarrow F(a) \tag{4.11}$$
$$F(g(a)) \rightarrow A \tag{4.12}$$

The real dependency graph will not contain any arrow, since there is no substitutions σ, τ such that $F(a)\sigma \xrightarrow[\mathcal{R}]{*} F(g(a))\tau$ or $F(a)\sigma \xrightarrow[\mathcal{R}]{*} A\tau$. However, the estimated dependency graph by unification will contain two arrows, since $REN(CAP(F(a))) = F(x)$ unifies with $F(g(a))$. The estimated dependency graph by unification is depicted in figure 4.3.

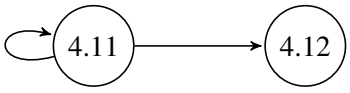

Figure 4.3: Dependency graph

On the other hand, the estimation by the tree automata will give exactly the same dependency graph as the real one. The estimated graph will not contain any arrow. For example, suppose we use the approximation mapping S. Then by Definition 4.3.7 we will have $(\mathcal{R}^{-1})_s = \{f(a) \rightarrow x, b \rightarrow x\}$. Hence the set of terms $(\xrightarrow[(\mathcal{R}^{-1})_s]{*})[\{F(a)\}]$ consists of all terms of the form $f^n(a), f^n(b), F(f^n(a)), F(f^n(b))$ with $n > 0$. But since the term $F(g(a))$ does not belong to this set, by Definition 4.3.8 there is no arrow in the estimated dependency graph by tree automata (no arrow from (4.11) to (4.12), and from (4.11) to (4.11)). Because the second condition of Definition 4.3.8 is violated. In this case, the estimated dependency graph by the tree automata is better than the estimation by unification.

However in general, the tree automata techniques are computationally expensive. In [Mid01], it is proposed that the approximations with the tree automata techniques should be tried only if the tools based on unification fail to prove termination. There are other possible ways to handle some arrows or cycles which actually do not exist in the real graph, besides refining techniques to estimate the dependency graph. That is, by introducing another processor which may permit us to ignore certain cycles or arrows in the estimated dependency graph. In the following, we will mention a DP processor called *subterm criterion processor* which exactly does this task.

4.3.2 Subterm Criterion Processor

This processor was first introduced by Hirokawa and Middeldorp in [HM04]. It tries to find an *argument projection* π that can project each terms $f^\sharp(t_1, \ldots, t_n)$ to one of its arguments $t_1, \ldots, t_n$. If for every dependency pair $l \rightarrow r$, the projected r is a subterm of the projected l, then all the dependency pairs $l \rightarrow r$ whose projections are in the strict subterm relation may be deleted.

Definition 4.3.10 (argument projections, subterm criterion processor)**.**

- An *argument projection* π for a DP problem $\mathcal{P}$, is a mapping that assigns to every dependency pair symbol $f^\sharp$ in $\mathcal{P}$, one of its argument positions. For an argument projection $\pi(f^\sharp)$, the extension to terms is defined by $\pi(f^\sharp(t_1, \ldots, t_n)) = t_{\pi(f^\sharp)}$.
- Let π be an argument projection for a DP problem $\mathcal{P}$. The *subterm criterion processor* is defined by
$$\phi_{SC}((\mathcal{P}, \mathcal{R})) = \begin{cases} (\{l \to r \in \mathcal{P} \mid \pi(l) = \pi(r)\}, \mathcal{R}) & \text{if } \pi(l) \trianglerighteq \pi(r) \text{ for every } l \to r \in \mathcal{P}, \\ (\mathcal{P}, \mathcal{R}) & \text{otherwise.} \end{cases}$$

Example 4.3.11.

We will continue simplifying the DP problems from Example 4.1.7 and Example 4.3.2. In the beginning, the initial DP problem consists of three dependency pairs:

$$\begin{aligned} (4.5)\ MINUS(s(x), s(y)) &\to MINUS(x, y) \\ (4.6)\ DIV(s(x), s(y)) &\to MINUS(x, y) \\ (4.7)\ DIV(s(x), s(y)) &\to DIV(minus(x, y), s(y)) \end{aligned}$$

It has been shown that proving finiteness of this DP problem can be decomposed into proving finiteness of two DP problems $(\{(4.5)\}, \mathcal{R})$, and $(\{(4.7)\}, \mathcal{R})$. By using the subterm criterion processor we can solve the first DP problem. We do this by projecting the MINUS symbol to its second argument. This will work since, by $\pi(MINUS) = 2$, the projected (4.5) are in the strict subterm relation:

$$\pi(MINUS(s(x), s(y))) = s(y) \rhd y = \pi(MINUS(x, y))$$

By the subterm criterion processor we can delete this pair, and obtain a finite DP problem $(\emptyset, \mathcal{R})$ from the first DP problem $(\{(4.5)\}, \mathcal{R})$. In this case, the second DP problem $(\{(4.7)\}, \mathcal{R})$ cannot be solved by the subterm criterion, since for every possible argument projection for DIV, the projected left and right -hand sides are not in the strict subterm relation.

Theorem 4.3.12. *The subterm criterion processor ϕ_{SC} is sound and complete.*

The basic idea why we can disregard the dependency pairs whose projected left and right -hand side are in the strict subterm relation is because, those dependency pairs cannot be applied infinitely often on some infinite DP chain. Consider a DP chain

$$s_1 \xrightarrow[\mathcal{R}]{*} t_1 \xrightarrow[\mathcal{P}]{} s_2 \xrightarrow[\mathcal{R}]{*} t_2 \xrightarrow[\mathcal{P}]{} \ldots \tag{4.13}$$

where $s_1, t_1, s_2, t_2 \ldots$ are terminating with respect to $\mathcal{R}$ (Definition 4.2.1). We know that for every $l \to r \in \mathcal{R}$, $l \trianglerighteq r$, we can transform this chain to

$$\pi(s_1) \xrightarrow[\mathcal{R}]{*} \pi(t_2) \trianglerighteq \pi(s_3) \xrightarrow[\mathcal{R}]{*} \pi(t_4) \trianglerighteq \pi(s_5) \xrightarrow[\mathcal{R}]{*} \ldots \tag{4.14}$$

In this chain, $\pi(t_i) \rhd \pi(s_{i+1})$ cannot occur infinitely often. Suppose this is the case. Then we can transform (4.14) into an infinite chain over $\xrightarrow[\mathcal{R}]{} \cup \rhd$ starting from $\pi(s_1)$. Since the relation

$\rhd$ is well-founded, the infinite sequence must contain infinitely many $\xrightarrow[\mathcal{R}]{}$ steps. Since $\rhd$ is commutative over $\xrightarrow[\mathcal{R}]{}$ (i.e. $\rhd \cdot \xrightarrow[\mathcal{R}]{} \subseteq \xrightarrow[\mathcal{R}]{} \cdot \rhd$), we can repeatedly use this property and obtain an infinite $\xrightarrow[\mathcal{R}]{}$ sequence starting from $\pi(s_1)$. This contradicts that $\pi(s_1)$ is terminating with respect to $\mathcal{R}$. Hence $\pi(t_i) \rhd \pi(s_{i+1})$ cannot occur infinitely often in (4.14). This fact can also be lifted to (4.13): dependency pairs whose projected left and right -hand sides are in the strict subterm relation cannot occur infinitely often in (4.13). Thus to prove that a DP problem is finite, in this case we can disregard the pairs whose projected left and right -hand sides are in the strict subterm relation.

The main problem of the application of this processor is that it can only be applied to very restricted DP problems. As it has been shown in Example 4.3.11 there is still one DP problem left that cannot be solved by this processor. In the following we will present another processor which can be used to handle this problem. This processor uses the approach of a pair of orders $(>, \gtrsim)$, and it is called the *reduction pair processor*.

4.3.3 Reduction Pair Processor

This processor is based on the technique of reduction orders. Let's recall the idea of the reduction orders from Chapter 2. In the reduction order method, we try to find a well-founded order $>$ such that for every reduction chain

$$t_1 \xrightarrow[\mathcal{R}]{} t_2 \xrightarrow[\mathcal{R}]{} t_3 \xrightarrow[\mathcal{R}]{} \dots \tag{4.15}$$

we have $t_1 > t_2 > t_3 > \dots$ In order to make the checking $s > t$ sufficient for the rewrite rules, we require $>$ to be closed under context and substitution. Hence termination of a TRS $\mathcal{R}$ is proved if we can find such order $>$ such that for every rewrite rules $l \to r \in \mathcal{R}$, $l > r$. Similar case also applies to prove finiteness of some DP problem. Instead of searching a single strict order, we may search for a pair of compatible strict and weak orders $>$ and $\gtrsim$, where $>$ is well-founded, such that for every DP chain

$$s_1 \xrightarrow[\mathcal{R}]{*} t_1 \xrightarrow[\mathcal{P}]{} s_2 \xrightarrow[\mathcal{R}]{*} t_2 \xrightarrow[\mathcal{P}]{} \dots$$

we have $s_1 \gtrsim t_1 > s_2 \gtrsim t_2 > \dots$ Since we also want to make the checking $s \gtrsim t$ and $t > s$ sufficient for the rewrite rules and the dependency pairs, we require $>$ to be closed under substitution, and $\gtrsim$ closed under context and substitution. Note that the order $>$ is not necessarily closed under context, since all of the reduction steps of $\mathcal{P}$ are done at the root positions. If we can find such pair of orders $(>, \gtrsim)$ such that for every $l \to r \in \mathcal{P}, l > r$, and for every $l \to r \in \mathcal{R}, l \gtrsim r$, then we can prove finiteness of the DP problem $(\mathcal{P}, \mathcal{R})$. We will call these pairs of orders *reduction pairs*.

Definition 4.3.13 (reduction pairs)**.**

A *reduction pair* is a pair $(>, \gtrsim)$, where the strict order $>$ is well-founded, closed under substitution, and compatible with the weak order $\gtrsim$ (i.e. $> \circ \gtrsim \subseteq >$ and $\gtrsim \circ > \subseteq >$), and $\gtrsim$ is closed under context and substitution.

Now we will formalize this idea in the context of the DP framework. The basic idea of the *reduction pair processor* is to delete some dependency pairs, if one can find a reduction pair $(>, \gtrsim)$ such that all the dependency pairs are decreasing with respect to either $>$ or $\gtrsim$, and all the rewrite rules are weakly decreasing with respect to $\gtrsim$. If such a reduction pair can be found, then the strictly decreasing dependency pairs can be deleted.

Definition 4.3.14 (reduction pair processor)**.**

- Let $\mathcal{P}$ be the set of rules and $>$ be some order relation. $\mathcal{P}_>$ is the set of rules of $\mathcal{P}$ that are decreasing with respect to the order relation $>$, i.e. $\mathcal{P}_> = \{l \to r \in \mathcal{P} \mid l > r\}$.
- Let $(>, \gtrsim)$ be a reduction pair. The *reduction pair processor* is defined by

$$\phi_{RP}((\mathcal{P}, \mathcal{R})) = \begin{cases} (\mathcal{P} \backslash \mathcal{P}_>, \mathcal{R}) & \text{if } \mathcal{P} = \mathcal{P}_> \cup \mathcal{P}_{\gtrsim} \text{ and } \mathcal{R} = \mathcal{R}_{\gtrsim} \\ (\mathcal{P}, \mathcal{R}) & \text{otherwise} \end{cases}$$

Example 4.3.15.

We will continue proving the finiteness of the DP problem from Example 4.3.2 and Example 4.3.11. This DP problem comes from the TRS of substraction and division:

$$\begin{aligned} minus(x, 0) &\to x && (4.16)\\ minus(s(x), s(y)) &\to minus(x, y) && (4.17)\\ div(0, s(y)) &\to 0 && (4.18)\\ div(s(x), s(y)) &\to s(div(minus(x, y), s(y))) && (4.19) \end{aligned}$$

Initially, the DP problem consists of three dependency pairs

$$\begin{aligned} (4.5)\ MINUS(s(x), s(y)) &\to MINUS(x, y)\\ (4.6)\ DIV(s(x), s(y)) &\to MINUS(x, y)\\ (4.7)\ DIV(s(x), s(y)) &\to DIV(minus(x, y), s(y)) \end{aligned}$$

As it has been shown before, we can decompose the initial DP problem $(\{(4.5), (4.6), (4.7)\}, \mathcal{R})$ to $(\{(4.5)\}, \mathcal{R})$ and $(\{(4.7)\}, \mathcal{R})$ by using the dependency graph processor. It also has been shown that we can prove finiteness of the first DP problem by using the subterm criterion processor. Now we will show that we can prove finiteness of the second DP problem by using the reduction pair processor. Consider the reduction pair $(>_\mathcal{P}, \geq_\mathcal{P})$ induced by a polynomial order $\mathcal{P}$ that can be seen as counting the function symbol s, i.e. $DIV_\mathcal{P}(x, y) = div_\mathcal{P}(x, y) = x + y$, $0_\mathcal{P} = 0$, and $s_\mathcal{P}(x) = x + 1$. By using these interpretations, all the rewrite rules are weakly decreasing and the dependency pairs (4.7) is strictly decreasing. Hence we can delete this dependency pair, and simplify the second DP problem $(\{(4.7)\}, \mathcal{R})$ to a finite DP problem $(\emptyset, \mathcal{R})$. Now all the simplified DP problems derived from the initial DP problem are finite. Since all the DP processors that we use are sound, we can conclude that the initial DP problem is also finite. Hence we successfully proved that the initial TRS is terminating by using the DP framework, where the termination proof is depicted in Figure 4.4.

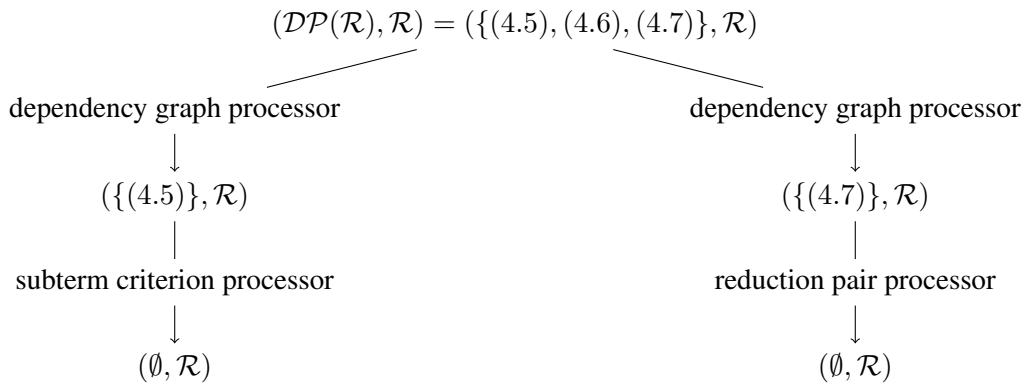

Figure 4.4: The dependency pair framework

Theorem 4.3.16. *The reduction pair processor is sound and complete [AG00].*

The basic idea why the reduction pair processor is correct is simple. Suppose we can find a reduction pair $(>, \gtrsim)$ such that $\mathcal{P} = \mathcal{P}_> \cup \mathcal{P}_\gtrsim$, $\mathcal{P}_> \neq \emptyset$, and $\mathcal{R} = \mathcal{R}_\gtrsim$. Then for every DP chain

$$s_1 \xrightarrow[\mathcal{R}]{*} t_1 \xrightarrow[\mathcal{P}]{} s_2 \xrightarrow[\mathcal{R}]{*} t_2 \xrightarrow[\mathcal{P}]{} \dots$$

the dependency pairs from $\mathcal{P}_>$ cannot be used infinitely often. Otherwise it will contradicts the well-foundedness of $>$, since $>$ and $\gtrsim$ are compatible. Hence in proving some DP problem is finite, we do not need to consider the strictly decreasing dependency pairs.

In the practical implementation, one can use the techniques available from the reduction orders method to search for a suitable reduction pair $(>, \gtrsim)$. For example, one may use polynomial interpretations, matrix interpretations, or simplification orders like RPOs and KBOs. There is also some practical improvement such that we do not need to show weak decreasingness for all rewrite rules. Instead, it is sufficient to show weak decreasingness only for *usable rules* ([GTSF06], [HM04]). The set of *usable rules* is a superset of those rules that may reduce the right-hand side of the dependency pairs, if their variables are instantiated with some normal forms.

Definition 4.3.17 (usable symbols, usable rules)**.**

- The set of *usable symbols* of a DP Problem $(\mathcal{P}, \mathcal{R})$ is $\mathcal{US}(\mathcal{P}, \mathcal{R}) = \bigcup_{l \to r \in \mathcal{P}} \mathcal{US}(r)$, where for a term t, the set $\mathcal{US}(t)$ is recursively defined as follows
  - If t is a variable, $\mathcal{US}(t) = \emptyset$.
  - If $t = f(t_1, \dots, t_n)$, $\mathcal{US}(t)$ is the least set such that
    - $f \in \mathcal{US}(t)$
    - $\mathcal{US}(t_1), \dots, \mathcal{US}(t_n) \subseteq \mathcal{US}(t)$.
    - if $l \to r \in \mathcal{R}$, and $root(l) \in \mathcal{US}(t)$, then $\mathcal{F}un(r) \subseteq \mathcal{US}(t)$.

- The set of *usable rules* of a DP Problem $(\mathcal{P}, \mathcal{R})$ is $\mathcal{U}(\mathcal{P}, \mathcal{R}) = \{l \to r \in \mathcal{R} \mid root(l) \in \mathcal{US}(\mathcal{P}, \mathcal{R})\}$.

Example 4.3.18.

For example, consider the DP problem $(\mathcal{P}, \mathcal{R})$ where $\mathcal{R}$ consists of three rewrite rules (written in infix notation),

$$x + 0 \to x \tag{4.20}$$
$$x \times 0 \to 0 \tag{4.21}$$
$$x \times s(y) \to (x + 0) \times s(y) \tag{4.22}$$

and $\mathcal{P}$ consists of two dependency pairs

$$x \times^\sharp s(y) \to (x + 0) \times^\sharp s(y) \tag{4.23}$$
$$x \times^\sharp s(y) \to x +^\sharp 0 \tag{4.24}$$

By Definition 4.3.17, the usable symbols here are $\times^\sharp$, $+^\sharp$, and $+$. Hence the only usable rule is (4.20).

By using the refined reduction pair processor, we only need to find a weak order that can orient the usable rules, instead of all rules. In the case of the DP problem from Example 4.3.18, we only need to make sure that (4.20) is weakly decreasing, and do not need to consider (4.21) and (4.22). However this weaker restriction only holds if the reduction pair satisfies some additional property called c_ϵ*-compatibility*.

Definition 4.3.19 (c_ϵ-compatible)**.**

The reduction pair $(>, \gtrsim)$ is c_ϵ*-compatible* if $c(x, y) \gtrsim x, y$ for some fresh symbol c.

We can reformulate the definition of the reduction pair processor from Definition 4.3.14 as a processor that can delete some dependency pairs, if we can find a c_ϵ*-compatible* reduction pair $(>, \gtrsim)$ such that all the dependency pairs are decreasing with respect to either $>$ or $\gtrsim$, and the usable rules are weakly decreasing with respect to $\gtrsim$. If such a c_ϵ-compatible reduction pair can be found, then the strictly decreasing dependency pairs can be deleted.

Definition 4.3.20 (reduction pair processor with usable rules)**.**

Let $(\mathcal{P}, \mathcal{R})$ be a DP problem and $(>, \gtrsim)$ be a c_ϵ-compatible reduction pair. The *reduction pair processor with usable rules* is defined by

$$\phi_{RPU}((\mathcal{P}, \mathcal{R})) = \begin{cases} (\mathcal{P} \backslash \mathcal{P}_>, \mathcal{R}) & \text{if } \mathcal{P} = \mathcal{P}_> \cup \mathcal{P}_\gtrsim \text{ and } \mathcal{U}(\mathcal{P}, \mathcal{R}) \subseteq \mathcal{R}_\gtrsim \\ (\mathcal{P}, \mathcal{R}) & \text{otherwise} \end{cases}$$

The detailed proof of why the restriction of c_ϵ-compatible on the reduction pairs may allow us to show weak decreasingness only for usable rules, is described in [GTSF06] and [HM04]. But the basic idea is because every DP chain

$$s_1 \xrightarrow[\mathcal{R}]{*} t_1 \xrightarrow[\mathcal{P}]{} s_2 \xrightarrow[\mathcal{R}]{*} t_2 \xrightarrow[\mathcal{P}]{} \dots \tag{4.25}$$

where $s_1, t_1, s_2, t_2 \ldots$ are terminating with respect to $\mathcal{R}$ (Definition 4.2.1), can be mapped by a special mapping called $\mathcal{I}_1$ to a DP chain that does not use the non-usable rules. In this mapped DP chain, the application of the non-usable rules are replaced by the application of the usable rules and the rules from TRS C_ϵ, $C_\epsilon = \{c(x,y) \to x, c(x,y) \to y\}$. The mapping of DP chain (4.25) by $\mathcal{I}_1$ can be illustrated as follows:

$$\mathcal{I}_1(s_1) \xrightarrow[\mathcal{U}(\mathcal{P},\mathcal{R})\cup C_\epsilon]{*} \mathcal{I}_1(t_1) \xrightarrow[C_\epsilon]{*} \cdot \xrightarrow[\mathcal{P}]{} \mathcal{I}_1(s_2) \xrightarrow[\mathcal{U}(\mathcal{P},\mathcal{R})\cup C_\epsilon]{*} \mathcal{I}_1(t_2) \xrightarrow[C_\epsilon]{*} \cdot \xrightarrow[\mathcal{P}]{} \ldots \tag{4.26}$$

The reason why we can replace the application of the non-usable rules with the application of the usable rules and the rules from C_ϵ lies in the definition of the mapping $\mathcal{I}_1$. The mapping $\mathcal{I}_1(t)$ intuitively "collects" all terms that can be obtained from t by reductions on or below the subterms with non-usable root symbols in zero or more steps.

Definition 4.3.21 (the mapping $\mathcal{I}_1$)**.**

Let $(\mathcal{P}, \mathcal{R})$ be a DP problem, where $\mathcal{F}$ the signature of $\mathcal{R}$ and $\mathcal{F}^\sharp$ the signature of $\mathcal{P}$. Let $\mathcal{US}(\mathcal{P}, \mathcal{R})$ be the set of usable symbols of $(\mathcal{P}, \mathcal{R})$ and t a term over $\mathcal{T}(\mathcal{F}^\sharp, \mathcal{V})$ that terminates with respect to $\mathcal{R}$. The interpretation $\mathcal{I}_1$ is a mapping from $\mathcal{T}(\mathcal{F}^\sharp, \mathcal{V})$ to terms in $\mathcal{T}(\mathcal{F}^\sharp \cup \{c, d\}, \mathcal{V})$ where c, d are fresh function symbols, and defined as follows

$$\mathcal{I}_1(t) = \begin{cases} t & \text{if } t \text{ is variable} \\ f(\mathcal{I}_1(t_1), \ldots, \mathcal{I}_1(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } f \in \mathcal{US}(\mathcal{P}, \mathcal{R}) \\ Comp(\{f(\mathcal{I}_1(t_1), \ldots, \mathcal{I}_1(t_n))\} \cup Red_1(t)) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } f \notin \mathcal{US}(\mathcal{P}, \mathcal{R}) \end{cases}$$

Here $Red_1(t) = \{\mathcal{I}_1(t') \mid t \underset{\mathcal{R}}{\to} t'\}$, and if we assume there is a total well-founded order on terms over $\mathcal{T}(\mathcal{F}^\sharp \cup \{c, d\}, \mathcal{V})$ and t is smaller than all terms in M, then $Comp(\{t\} \uplus M) = c(t, Comp(M))$, and $Comp(\emptyset) = d$.

Example 4.3.22 ([HM04])**.**

We will give some example of the application of the mapping $\mathcal{I}_1$. We will map a DP chain that involves the application of the non-usable rules, to a DP chain that only use the usable rules and the rules from TRS C_ϵ. Consider a DP chain of the DP problem from Example 4.3.18

$$\begin{aligned} & ((0+0) \times 0) \times^\sharp s(0) && (4.27) \\ \xrightarrow[\mathcal{P}]{(4.23)} \quad & (((0+0) \times 0) + 0) \times^\sharp s(0) \\ \xrightarrow[\mathcal{R}]{(4.20)} \quad & ((0+0) \times 0) \times^\sharp s(0) \\ \xrightarrow[\mathcal{R}]{(4.21)} \quad & 0 \times^\sharp s(0) \\ \xrightarrow[\mathcal{P}]{(4.23)} \quad & (0+0) \times^\sharp s(0) \\ \xrightarrow[\mathcal{R}]{(4.20)} \quad & 0 \times^\sharp s(0) \\ \xrightarrow[\mathcal{P}]{(4.23)} \quad & \ldots \end{aligned}$$

This DP chain involves the application of the non-usable rules (4.21). We will show that we can map this chain by $\mathcal{I}_1$ to a chain that instead of using (4.21) uses the usable rule (4.20), and rules from C_ϵ . For this purpose first we map the first term $((0+0)\times 0)\times^\sharp s(0)$ by precomputing $\mathcal{I}_1(0\times 0)$ and $\mathcal{I}_1((0+0)\times 0)$.

$$\begin{aligned}
\mathcal{I}_1(0\times 0) &= c(0\times 0, c(0,d)).\\
\mathcal{I}_1((0+0)\times 0) &= c((0+0)\times 0, c(c(0\times 0, c(0,d)), c(0,d))).\\
\mathcal{I}_1(((0+0)\times 0)\times^\sharp s(0)) &= c((0+0)\times 0, c(c(0\times 0, c(0,d)), c(0,d)))\times^\sharp s(0)
\end{aligned}$$

Now we can map (4.27) by $\mathcal{I}_1$ to

$$\begin{aligned}
& c((0+0)\times 0, c(c(0\times 0, c(0,d)), c(0,d)))\times^\sharp s(0)\\
\xrightarrow[\mathcal{P}]{(4.23)} \quad & c((0+0)\times 0, c(c(0\times 0, c(0,d)), c(0,d))) + 0\times^\sharp s(0)\\
\xrightarrow[\mathcal{U}(\mathcal{P},\mathcal{R})]{(4.20)} \quad & c((0+0)\times 0, c(c(0\times 0, c(0,d)), c(0,d)))\times^\sharp s(0)\\
\xrightarrow[\mathcal{C}_\epsilon]{+} \quad & 0\times^\sharp s(0)\\
\xrightarrow[\mathcal{P}]{(4.23)} \quad & (0+0)\times^\sharp s(0)\\
\xrightarrow[\mathcal{U}(\mathcal{P},\mathcal{R})]{(4.20)} \quad & 0\times^\sharp s(0)\\
\xrightarrow[\mathcal{P}]{(4.23)} \quad & \ldots
\end{aligned}$$

The possibility of this mapping shows that for proving finiteness of the DP problem $(\mathcal{P},\mathcal{R})$ it is sufficient to prove finiteness of the DP problem $(\mathcal{P}, \mathcal{U}(\mathcal{P},\mathcal{R})\cup C_\epsilon)$. If we restrict the reduction order to be c_ϵ-compatible, then weak decreasingness is only required for the usable rules instead of all rules. In practice, one can use a polynomial interpretation that interprets the new function symbol c as the addition of its two arguments, $c_\mathcal{P}(x,y) = x+y$. Such polynomial order satisfies the c_ϵ-compatible property. We can also use simplification orders, since they are also c_ϵ-compatible, because of the subterm property.

Besides the refinement by using the usable rules, there is also another refinement for the reduction pair processor. Recall that in the definition of the reduction pair $(>,\gtrsim)$ (Definition 4.3.14), we do not require the strict order to be closed under context. Hence in this case, we may use a polynomial interpretation that is not strictly monotone. We may use some polynomials that only depends on some of its arguments. The same case applies for the simplification orders. We may eliminate some of the arguments first before searching for suitable simplification orders. We can formulate this idea of refinement by incorporating the notion of *argument filtering*.

Definition 4.3.23 (argument filtering)**.**

An *argument filtering* is a mapping π such that for every n-ary function symbol f, $\pi(f) = [i_1,\ldots,i_m]$ where $1\le i_1 < \ldots < i_m \le n$. The extension to terms is defined by

$$\pi(t) = \begin{cases} t & \text{if } t \text{ is a variable}\\ f(\pi(t_{i_1}),\ldots,\pi(t_{i_m})) & \text{if } t = f(t_1,\ldots,t_n) \text{ and } \pi(f) = [i_1,\ldots,i_m]\end{cases}$$

The extension to TRSs is defined by $\pi(\mathcal{R}) = \{\pi(l) \to \pi(r) \mid l \to r \in \mathcal{R}\}$.

Moreover, we also have the notation $>_\pi$ to denote the order over some argument filtering π. We say that s is decreasing to t with respect to π, i.e. $s >_\pi t$, if and only if the filtered s is decreasing to the filtered t, i.e. $\pi(s) > \pi(t)$.

Definition 4.3.24.
Let $>$ be an order relation and π an argument filtering. For every terms s, t, we have $s >_\pi t$ if and only if $\pi(s) > \pi(t)$. It is shown in [AG00] that if $(>, \gtrsim)$ is a reduction pair then $(>_\pi, \gtrsim_\pi)$ is also a reduction pair.

Now let us consider again the idea of the reduction pair processor with the usable rules. In this refinement, we only need to show weak decreasingness for the usable rules instead of all rules. In fact, by incorporating the notion of argument filtering, we can even reduce the number of the usable rules that is needed to be shown weakly decreasing. This is so since the argument filtering may eliminate some usable symbols from the right-hand sides of the dependency pairs and the rewrite rules. For this purpose, we again define the set of usable symbols and usable rules, but now with respect to some argument filtering. This set at least will be smaller than or equal to the previous set of usable symbols and usable rules without argument filtering.

Definition 4.3.25 (usable symbols and usable rules with respect to some argument filtering)**.**

- The set of *usable symbols with respect to the argument filtering* π of the DP Problem $(\mathcal{P}, \mathcal{R})$ is $\mathcal{US}_\pi(\mathcal{P}, \mathcal{R}) = \bigcup_{l \to r \in \mathcal{P}} \mathcal{US}_\pi(r)$, where for a term t, the set $\mathcal{US}_\pi(t)$ is recursively defined as follows
  - If t is a variable, $\mathcal{US}_\pi(t) = \emptyset$.
  - If $t = f(t_1, \ldots, t_n)$, $\mathcal{US}_\pi(t)$ is the least set such that
    - $f \in \mathcal{US}_\pi(t)$
    - if $\pi(f) = [i_1, \ldots, i_m]$, then $\mathcal{US}_\pi(t_{i_1}) \subseteq \mathcal{US}_\pi(t), \ldots, \mathcal{US}_\pi(t_{i_m}) \subseteq \mathcal{US}_\pi(t)$.
    - if $l \to r \in \mathcal{R}$ and $root(l) \in \mathcal{US}_\pi(t)$, then $\mathcal{F}un(r) \subseteq \mathcal{US}_\pi(t)$.
- The set of *usable rules with respect to an argument filtering* π of a DP Problem $(\mathcal{P}, \mathcal{R})$ is $\mathcal{U}_\pi(\mathcal{P}, \mathcal{R}) = \{l \to r \in \mathcal{R} \mid root(l) \in \mathcal{US}_\pi(\mathcal{P}, \mathcal{R})\}$.

Now we can define the reduction pair processor that deletes some dependency pairs from $\mathcal{P}$, if we can find an argument filtering π and a c_ϵ-*compatible* reduction pair $(>, \gtrsim)$, such that all the dependency pairs are either decreasing with respect to $>_\pi$ or $\gtrsim_\pi$, and the *usable rules with respect to the argument filtering* π are weakly decreasing with respect to $\gtrsim_\pi$. If we can find such argument filtering and c_ϵ-compatible reduction pair, then we can delete the dependency pairs that are strictly decreasing w.r.t. $>_\pi$.

Definition 4.3.26 (reduction pair processor with usable rules and argument filtering)**.**

Let $(\mathcal{P}, \mathcal{R})$ be a DP problem, $(>, \gtrsim)$ a c_ϵ-compatible reduction pair, and π an argument filtering. The *reduction pair processor with usable rules and argument filtering* is defined by

$$\phi_{RPUA}((\mathcal{P}, \mathcal{R})) = \begin{cases} (\mathcal{P} \backslash \mathcal{P}_{>_\pi}, \mathcal{R}) & \text{if } \mathcal{P} = \mathcal{P}_{>_\pi} \cup \mathcal{P}_{\gtrsim_\pi} \text{ and } \mathcal{U}_\pi(\mathcal{P}, \mathcal{R}) \subseteq \mathcal{R}_{\gtrsim_\pi} \\ (\mathcal{P}, \mathcal{R}) & \text{otherwise} \end{cases}$$

Theorem 4.3.27. *The reduction pair processor with usable rules and argument filtering is sound and complete.*

The complete proof of this theorem is described in [GTSF06]. But the basic idea is similar to the previous one. We can map every infinite DP chain over $(\mathcal{P}, \mathcal{R})$ to an infinite DP chain over $(\mathcal{P}, \mathcal{U}_\pi(\mathcal{P}, \mathcal{R}) \cup C_\epsilon)$. But now we do it with the mapping $\mathcal{I}_2$, a similar mapping like the previous one: $\mathcal{I}_1$. We will show the different of $\mathcal{I}_1$ and $\mathcal{I}_2$ in the following example.

Example 4.3.28.

Consider a DP problem $(\mathcal{P}, \mathcal{R})$ where the set of the usable symbols is $\mathcal{US}_\pi(\mathcal{P}, \mathcal{R}) = \{:, rev_2\}$. Let t be a term $t = rev_1(x, l) : rev_2(x, l)$, and π an argument filtering that eliminates the first argument of the function symbols : and rev_2, i.e. $\pi(:) = \pi(rev_2) = [2]$. If we apply the mapping $\mathcal{I}_1$ to the term t, then we obtain $\mathcal{I}_1(t) = c(rev1(x, l), d) : rev2(x, l)$. But if we apply the mapping $\mathcal{I}_2$ to the term t, then we obtain $\mathcal{I}_2(t) = : rev_2(l)$. So, the mapping $\mathcal{I}_2$ is basically the same as the mapping $\mathcal{I}_1$, but it incorporates the argument filtering π.

Example 4.3.29.

We will show the advantage of the reduction pair processor that incorporates the argument filtering and the usable rules [GTSF06]. Consider the DP problem $(\mathcal{DP}(\mathcal{R}), \mathcal{R})$ from the TRS of the list reversal,

$$\begin{aligned} rev(nil) &\to nil \\ rev_1(x, nil) &\to x \\ rev_2(x, nil) &\to nil \\ rev(x : l) &\to rev_1(x, l) : rev_2(x, l) \\ rev_1(x, y : l) &\to rev_1(y, l) \\ rev_2(x, y : l) &\to rev(x : rev(rev_2(y, l))) \end{aligned}$$

The set of the dependency pairs, $\mathcal{DP}(\mathcal{R})$, consists of

$$REV(x : l) \to REV_1(x, l) \tag{4.28}$$
$$REV(x : l) \to REV_2(x, l) \tag{4.29}$$
$$REV_1(x, y : l) \to REV_1(y, l) \tag{4.30}$$
$$REV_2(x, y : l) \to REV(x : rev(rev_2(y, l))) \tag{4.31}$$
$$REV_2(x, y : l) \to REV(rev_2(y, l)) \tag{4.32}$$
$$REV_2(x, y : l) \to REV_2(y, l) \tag{4.33}$$

To prove the finiteness of the DP problem $(\mathcal{DP}(\mathcal{R}), \mathcal{R})$, first we may apply the dependency graph processor. The estimated dependency graph by unification is depicted in Figure 4.5. The

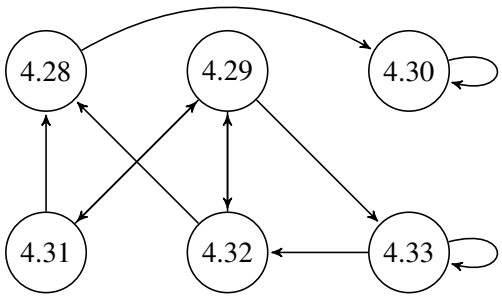

Figure 4.5: Dependency graph

dependency graph contains two SCCs, $\mathcal{S}_1 = \{(4.30)\}$ and $\mathcal{S}_2 = \{(4.29), (4.31), (4.32), (4.33)\}$. The first DP problem $(\mathcal{S}_1, \mathcal{R})$ can be handled by the subterm criterion processor. But the second DP problem cannot be handled with either the subterm criterion processor or the reduction pair processors without usable rules and argument filtering. Suppose we want to solve the second DP problem $(\mathcal{S}_2, \mathcal{R})$ with the reduction pair processor (Definition 4.3.14) or the reduction pair processor with usable rules (Definition 4.3.17). Then we need to find a reduction pair $(>, \gtrsim)$ that satisfies these constraints

$$
\begin{array}{rcl@{\qquad}rcl}
rev(nil) & \gtrsim & nil & rev(x:l) & \gtrsim & rev_1(x,l) : rev_2(x,l) \\
rev_1(x, nil) & \gtrsim & x & rev_1(x, y:l) & \gtrsim & rev_1(y,l) \\
rev_2(x, nil) & \gtrsim & nil & rev_2(x, y:l) & \gtrsim & rev(x : rev(rev_2(y,l)))
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
REV(x:l) & \stackrel{?}{>} & REV_2(x,l) & REV_2(x,y:l) & \stackrel{?}{>} & REV(rev_2(y,l)) \\
REV_2(x,y:l) & \stackrel{?}{>} & REV(x : rev(rev_2(y,l))) & REV_2(x,y:l) & \stackrel{?}{>} & REV_2(y,l)
\end{array}
$$

Note that the refinement by usable rules without argument filtering here is useless, since in this DP problem all the rewrite rules are also usable. We still need to show weak decreasingness for all the rewrite rules. It is shown in [GTSF03], that the standard reduction pair methods induced by RPOs, KBOs, or some restricted polynomial orders, cannot satisfy these constraints. In contrast, if we use the reduction pair processor with usable rules and argument filtering (Definition 4.3.26), then we can simplify these constraints and find a suitable reduction pair easily.

Consider the argument filtering π that eliminates the first argument of the function symbols

:, REV_2, and rev_2 i.e. $\pi(:) = \pi(REV_2) = \pi(rev_2) = [2]$. The filtered rewrite rules are

$$\begin{aligned} rev(nil) &\to nil && (4.34)\\ rev_1(x, nil) &\to x && (4.35)\\ rev_2(nil) &\to nil && (4.36)\\ rev(: l) &\to : rev_2(l) && (4.37)\\ rev_1(: l) &\to rev_1(y, l) && (4.38)\\ rev_2(: l) &\to rev(: rev(rev_2(l))) && (4.39) \end{aligned}$$

and the filtered dependency pairs are

$$\begin{aligned} (4.29)\ REV(: l) &\to REV_2(l)\\ (4.31)\ REV_2(: l) &\to REV(: rev(rev_2(l)))\\ (4.32)\ REV_2(: l) &\to REV(rev_2(l))\\ (4.33)\ REV_2(: l) &\to REV_2(l) \end{aligned}$$

Now the number of the usable rules is reduced. The rules (4.35) and (4.38) are no longer usable since the function symbols rev_1 is no longer usable. Hence by using the concept of argument filtering, we are not only eliminating the arguments and improve the search, but we also reduce the constraints for the usable rules. In this case, instead of searching for orders that satisfied the previous constraints, we only need to search for orders that satisfy these simplified constraints:

$$\begin{aligned} rev(nil) &\gtrsim nil & \qquad rev(: l) &\gtrsim : rev_2(l)\\ rev_2(nil) &\gtrsim nil & \qquad rev_2(: l) &\gtrsim rev(: rev(rev_2(l))) \end{aligned}$$

$$\begin{aligned} REV(: l) &\overset{?}{>} REV_2(l) & \qquad REV_2(: l) &\overset{?}{>} REV(rev_2(l))\\ REV_2(: l) &\overset{?}{>} REV(: rev(rev_2(l))) & \qquad REV_2(: l) &\overset{?}{>} REV_2(l) \end{aligned}$$

We can find easily some reduction pair that will satisfy these constraints. Consider the order that counts the occurrences of function symbol :, i.e. induced by the polynomial interpretations $:_{\mathbb{N}}(x) = x+1$, and $f_{\mathbb{N}}(x) = x$ for other unary function symbol f, and $nil_{\mathbb{N}} = 0$. This order will successfully orient the rewrite rules and the dependency pairs, and deletes all the dependency pairs except (4.31). However (4.31) can be deleted by the application of the dependency graph processor in the next step. At the end, we will have that all the DP problems derived from the initial DP problem are finite. Hence we can conclude that the initial DP problem is also finite because we use sound DP processors. So we have proved that the original TRS is terminating by using the DP framework. Its termination proof is depicted in Figure 4.6.

In the practical implementation, suitable argument filterings π are determined first before searching for the reduction pair $(>, \gtrsim)$. However, the number of the possible argument filterings is exponential in the number and in the arities of the function symbols. Some efficient search techniques for a suitable argument filtering is given in the paper [GTSF06]. In the next

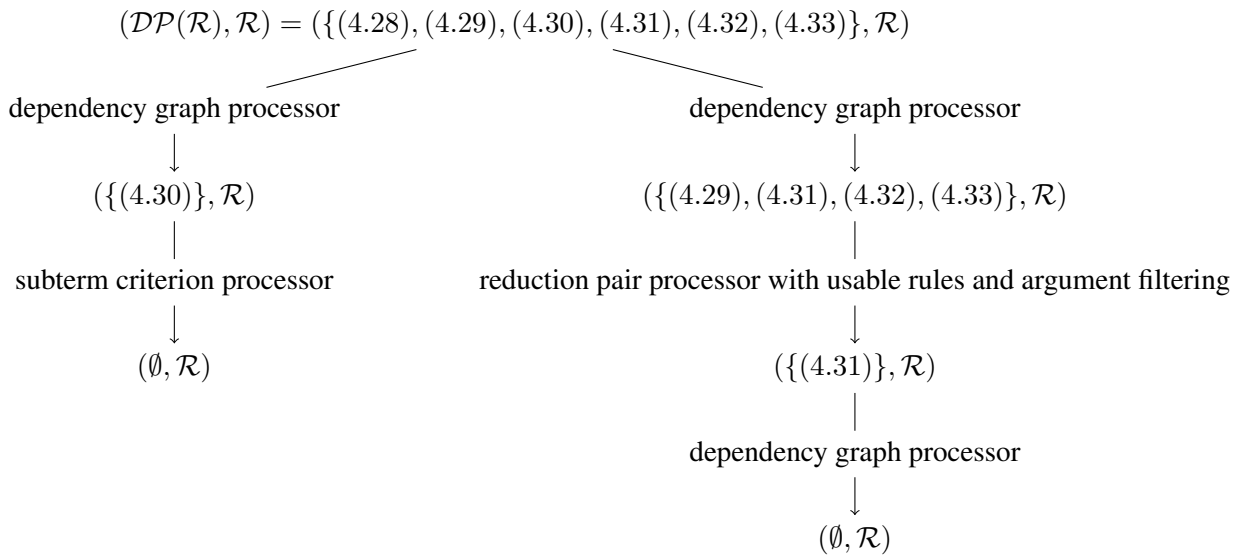

Figure 4.6: The dependency pair framework

subsection, we will discuss the processor that can also remove some rewrite rules. This processor is called *rule removal processor*.

4.3.4 Rule Removal Processor

Let's recall the reduction pair processor we have mentioned before. The reduction pair processor deletes some dependency pairs that are strictly decreasing, if we can find a reduction pair $(>, \gtrsim)$ that can orient the dependency pairs and the rewrite rules. Slightly different from the reduction pair processor, the rule removal processor also searches for a reduction pair, but it may delete some rewrite rules that are strictly decreasing. As a consequence, we have to add an additional restriction to the reduction pair $(>, \gtrsim)$. We require $>$ to be closed under context, and we are not allowed to use the techniques of argument filtering.

Definition 4.3.30 (rule removal processor)**.**
Let $(>, \gtrsim)$ be a reduction pair where $>$ is closed under context. The *rule removal processor* is defined by

$$\phi_{RR}((\mathcal{P}, \mathcal{R})) = \begin{cases} (\mathcal{P} \backslash \mathcal{P}_>, \mathcal{R} \backslash \mathcal{R}_>) & \text{if } \mathcal{P} = \mathcal{P}_> \cup \mathcal{P}_\gtrsim \text{ and } \mathcal{R} = \mathcal{R}_> \cup \mathcal{R}_\gtrsim \\ (\mathcal{P}, \mathcal{R}) & \text{otherwise} \end{cases}$$

Theorem 4.3.31. *The rule removal processor ϕ_{RR} is sound and complete.*

The basic idea of this processor is also simple. Suppose we can find a reduction pair $(>, \gtrsim)$ such that $\mathcal{P} = \mathcal{P}_> \cup \mathcal{P}_\gtrsim$, $\mathcal{R} = \mathcal{R}_> \cup \mathcal{R}_\gtrsim$, and $\mathcal{P}_>, \mathcal{R}_> \neq \emptyset$. Then for every DP chain

$$s_1 \xrightarrow[\mathcal{R}]{*} t_1 \xrightarrow[\mathcal{P}]{} s_2 \xrightarrow[\mathcal{R}]{*} t_2 \xrightarrow[\mathcal{P}]{} \dots$$

the dependency pairs and the rewrite rules from $\mathcal{P}_>$ and $\mathcal{R}_>$ cannot be used infinitely often. Otherwise it will contradicts the well-foundedness of $>$, since $>$ and $\gtrsim$ are compatible. Hence we can prove finiteness of some DP problem without considering the strictly decreasing rewrite rules and dependency pairs.

Example 4.3.32.

To illustrate the application of the rule removal processor, consider the DP problem

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

This DP problem is finite since intuitively, the consecutive applications of the dependency pair and the rewrite rule always reduce to smaller terms. However in this case it is impossible to show finiteness by using the reduction pair processor. There is no well-founded strict order $>$ that is closed under substitution that can orient the dependency pair. But fortunately, adding the property closed under context to the strict order $>$, will allow us to delete the rewrite rule, which in this case is very helpful. Consider the reduction pair induced by the strictly monotone polynomials: $MINUS_{\mathcal{P}}(x, y) = x + y$, $p_{\mathcal{P}}(x) = x$, and $s_{\mathcal{P}}(x) = x + 1$. The dependency pair is weakly decreasing with respect to this order, and the rewrite rule is strictly decreasing. The rule removal processor deletes the rewrite rule and we obtain a DP problem

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

which can be shown finite by the dependency graph processor.

The next DP processor that we will introduce is not based on searching for orders, but based on the technique to refine the dependency graph by *narrowing* [AG00] . The narrowing processor simplifies the DP problem $(\mathcal{P}, \mathcal{R})$ by transforming some dependency pairs $s \to t$ from $\mathcal{P}$ into several *narrowed* dependency pairs, such that showing finiteness of the transformed DP problem become easier. For example, after applying the narrowing processor, some DP processor can now be used to simplify the DP problems. We will give an example about this in the following subsection.

4.3.5 Narrowing Processor

Let's recall the idea of proving finiteness by finding some pair of orders. To prove that a DP problem is finite, we try to find a pair of compatible strict and weak orders $>$ and $\gtrsim$, where $>$ is well-founded, such that every DP chain

$$s_1 \xrightarrow[\mathcal{P}]{} t_1 \xrightarrow[\mathcal{R}]{*} s_2 \xrightarrow[\mathcal{P}]{} t_2 \xrightarrow[\mathcal{R}]{*} \dots \tag{4.40}$$

will imply $s_1 > t_1 \gtrsim s_2 > t_2 \gtrsim \dots$ Now consider two consecutive reduction steps $\xrightarrow[\mathcal{P}]{}$ on this chain, for example $s_1 \xrightarrow[\mathcal{P}]{} t_1$ and $s_2 \xrightarrow[\mathcal{P}]{} t_2$. The basic idea of *narrowing* is, if the reduction sequence from t_1 to s_2 in (4.40) is done in at least one step, i.e

$$t_1 \xrightarrow[\mathcal{R}]{} t'_1 \xrightarrow[\mathcal{R}]{*} s_2 \tag{4.41}$$

for some term t'_1, then instead of requiring $s_1 > t_1 \gtrsim s_2$ we may require $s_1 > t'_1 \gtrsim s_2$. We call the term t'_1 the *narrowed term* of t_1.

We can lift this idea in the context of the dependency pairs. For example consider the dependency pair that is used in the reduction step $s_1 \xrightarrow[\mathcal{P}]{} t_1$. Suppose $s \to t \in \mathcal{P}$. Then instead of requiring $s \to t$ to be strictly decreasing, we may compute the new dependency pair $s' \to t'$ that simulates the reduction sequence

$$s_1 \xrightarrow[\mathcal{P}]{} t_1 \xrightarrow[\mathcal{R}]{} t'_1$$

and require the new dependency pair $s' \to t'$ to be strictly decreasing. To compute the new dependency pair we can use again the definition of *narrowing*. In this case the left-hand side of the new dependency pair t' is the narrowed term of the left-hand side of the original dependency pairs t.

Definition 4.3.33 ($\mathcal{R}$-narrowing)**.**

A term t' is called *$\mathcal{R}$-narrowing of t with mgu μ* if a non-variable subterm of t: $t\mid_p$, unifies with the left-hand side of the rule $l \to r$ in $\mathcal{R}$, with mgu μ and $t' = t[r]_p\mu$.

For some dependency pairs $s \to t$, instead of only narrowing the term t, the substitutions derived from narrowing t should also be applied to the right-hand side s. Hence if $t_1, \ldots t_n$ are all possible narrowings of t by some substitutions $\mu_1, \ldots, \mu_n$, then proving that $s \to t$ is strictly decreasing is equivalent to proving $s\mu_1 \to t_1$, $s\mu_2 \to t_2$, ... or $s\mu_n \to t_n$ are strictly decreasing. We called these new pairs the *narrowed pairs* of $s \to t$.

The basic idea of the narrowing processor is to replace some dependency pairs $s \to t$ with their narrowed pairs. To make sure there is at least one step of narrowing from t, we require t to be not unifiable with any other left-hand sides of the dependency pairs. We also require t to be linear. But the restriction of linearity can be eliminated if we deal with the *innermost termination* case. However, we will not comment further about this case, and keep our focus on the general termination case.

Definition 4.3.34 (narrowing processor)**.**

Let $\mathcal{P}' = \mathcal{P} \cup \{s \to t\}$. The *narrowing processor* is defined by

$$\phi_N((\mathcal{P}', \mathcal{R})) = \begin{cases} (\mathcal{P} \cup \{s\mu_1 \to t_1, \ldots, s\mu_n \to t_n\}, \mathcal{R}) & \text{if } t_1, \ldots, t_n \text{ are all } \mathcal{R}\text{-narrowings of } t \text{ with} \\ & \text{mgu's } \mu_1, \ldots, \mu_n, \, t \text{ does not unify with the} \\ & \text{left-hand sides of pairs in } \mathcal{P}', \text{ and } t \text{ is linear.} \\ (\mathcal{P} \cup \{s \to t\}, \mathcal{R}) & \text{otherwise} \end{cases}$$

Example 4.3.35.

For example, consider the following DP problem $(\mathcal{P}, \mathcal{R})$, where $\mathcal{R}$ consists of

$$\begin{aligned}
min(x, 0) &\to x && (4.42)\\
min(s(x), s(y)) &\to min(x, y) && (4.43)\\
div(0, s(y)) &\to 0 && (4.44)\\
div(s(x), s(y)) &\to s(div(min(x, y), s(y))) && (4.45)\\
le(0, y) &\to true && (4.46)\\
add(0, y) &\to y && (4.47)\\
add(s(x), y) &\to s(add(x, y)) && (4.48)\\
add(min(x, s(0)), min(y, s(s(z)))) &\to add(min(y, s(s(z))), min(x, s(0))) && (4.49)
\end{aligned}$$

and $\mathcal{P}$ contains the dependency pair from the commutativity rule

$$ADD(min(x, s(0)), min(y, s(s(z)))) \to ADD(min(y, s(s(z))), min(x, s(0))) \quad (4.50)$$

In this case, the application of the narrowing processor will accommodate the application of other DP processors. For example, here initially we cannot apply the reduction pair processor to delete the dependency pair (4.50). There is no well-founded strict order that is closed under substitution that can orient this pair. However by applying the narrowing processor first, we can find such an order that can orient the narrowed pairs, and hence delete (4.50). Consider the rewrite rule (4.43). This is the only rule that can narrow the left-hand side of the dependency pair (4.50). We can use this rule to narrow the left-hand side of (4.50) in two different ways. The first narrowing is done at the first argument after substituting y with $s(y)$. From this narrowing we obtain $ADD(min(y, s(z)), min(x, s(0)))$. The second one is done at the second argument after substituting x with $s(x)$. From this one we obtain the term $ADD(min(y, s(s(z))), min(x, 0))$. Hence we have two narrowed dependency pairs of (4.50)

$$\begin{aligned}
ADD(min(x, s(0)), min(s(y), s(z))) &\to ADD(min(y, s(z)), min(x, s(0)))\\
ADD(min(s(x), s(0)), min(y, s(s(z)))) &\to ADD(min(y, s(s(z))), min(x, 0))
\end{aligned}$$

These two narrowed dependency pairs can be deleted easily by the reduction pair processor that counts the function symbols, i.e. a polynomial order $>_{\mathcal{P}}$ induced by the interpretations: $ADD_{\mathcal{P}}(x, y) = min_{\mathcal{P}}(x, y) = x + y + 1$, $s_{\mathcal{P}}(x) = x + 1$, and $0_{\mathcal{P}} = 1$. Hence at the end we will obtain a finite DP problem $(\emptyset, \mathcal{R})$, and prove termination of the original TRS $\mathcal{R}$.

The narrowing processor is sound. But in practice, the narrowing process may be applied infinitely many times. For instance, consider the TRS $\mathcal{R}$ from Example 4.1.7, with the dependency pair

$$DIV(s(x), s(y)) \to DIV(minus(x, y), s(y)) \quad (4.51)$$

We can obtain infinitely many applications of the narrowing processor to the dependency pair (4.51) by using the minus rules:

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

The DP problem $(\{(4.51)\}, \mathcal{R})$ may be narrowed to the DP problem $(\{(4.52), (4.53)\}, \mathcal{R})$,

$$DIV(s(x), s(0)) \rightarrow DIV(x, s(0)) \quad (4.52)$$
$$DIV(s(s(x)), s(s(y))) \rightarrow DIV(minus(x, y), s(s(y))) \quad (4.53)$$

Then in the next step we may narrow again (4.53), and obtain the DP problem $(\{(4.52), (4.54), (4.55)\}, \mathcal{R})$,

$$DIV(s(s(x)), s(s(0))) \rightarrow DIV(x, s(s(0))) \quad (4.54)$$
$$DIV(s(s(s(x))), s(s(s(y)))) \rightarrow DIV(minus(x, y), s(s(s(y)))) \quad (4.55)$$

Even though the DP problems are finite and the TRS is terminating, the narrowing processor may be applied infinitely often. To handle this problem, in the paper [GTSF06], a restriction for a *safe* transformation that guarantes that the application of the narrowing processor will terminate is developed.

The narrowing processes are *safe* if they reduce the number of pairs in the SCCs of the estimated dependency graph. We also call it safe if all the narrowed pairs disappear from the SCCs, i.e. the original dependency pairs are decreasing after the narrowing processes. The formal definition of a safe transformation will be given in the following definition. Let $o(s \rightarrow t)$ denote the set of the original dependency pairs whose repeated narrowing processes leads to the dependency pair $s \rightarrow t$.

Definition 4.3.36 (Safe narrowing transformations)**.**

Let $\mathcal{P} = (\mathcal{P}_1, \mathcal{R}_1)$ and $\mathcal{Q} = (\mathcal{P}_2, \mathcal{R}_2)$ be DP problems, and $\mathcal{Q}$ is the result after some narrowing processes of $\mathcal{P}$. The processes are *safe* if

- $\mid \cup_{\mathcal{S} \in SCC(\mathcal{P})} \mathcal{S} \mid > \mid \cup_{\mathcal{S} \in SCC(\mathcal{Q})} \mathcal{S} \mid$, or
- $\{o(s \rightarrow t) \mid s \rightarrow t \in \cup_{\mathcal{S} \in SCC(\mathcal{P})} \mathcal{S}\} \supset \{o(s \rightarrow t) \mid s \rightarrow t \in \cup_{\mathcal{S} \in SCC(\mathcal{Q})} \mathcal{S}\}$

According to this definition, the narrowing processes in the example above was not *safe*, but the narrowing processes in Example 4.3.35 was safe, since it can delete the original dependency pairs. Applying the narrowing processor in these *safe* case, is indeed successful in practice [GTSF06].

There is also another DP processor that can transform a dependency pair into several pairs such that finiteness problem become easier to solve. This processor is also based on the technique to refine the dependency graph [GTSF06], and it incorporates the function CAP and REN that were already introduced in the section of the dependency graph processor.

4.3.6 Instantiation Processor

Let's recall again the reduction pair methods. We prove that some DP problem is finite by finding a reduction pair $(>, \gtrsim)$ such that every rewrite rules $l \rightarrow r$ are weakly decreasing, i.e. $l \gtrsim r$, and every dependency pair $s \rightarrow t$ is strictly decreasing, i.e. $s > t$. However instead of showing directly the dependency pairs $s \rightarrow t$ are strictly decreasing, sometimes it is sufficient to show strictly decreasing only for some instances of them, e.g. $s\mu_1 \rightarrow t\mu_1 \ldots s\mu_n \rightarrow t\mu_n$. This is the case if we know already what is the form of terms that will be reduced by $s \rightarrow t$ in some DP

chain. We can estimate this form by checking which dependency pairs $u \to v$ can occur before $s \to t$, and then check whether $REN(CAP(v))$ and s are unifiable. If $REN(CAP(v))$ and s are unifiable with mgu μ, then $REN(CAP(v))\mu$ or $s\mu$ estimates the form of terms that will be reduced by $s \to t$ in some DP chain. Suppose $v_1, \ldots v_n$ are all the right-hand sides of the dependency pairs such that s unifies with $REN(CAP(v_1))$, ..., $REN(CAP(v_n))$ with mgu $\mu_1, \ldots, \mu_n$ respectively. Then instead of showing strict decreasingness for $s \to t$, it is sufficient to show it for the instances: $s\mu_1 \to t\mu_1, \ldots, s\mu_n \to t\mu_n$. The basic idea of the instantiation processor is to replace some dependency pairs $s \to t$ with its instances $s\mu_1 \to t\mu_1, \ldots, s\mu_n \to t\mu_n$. This idea is useful, since restricting the focus to solve the instantiated dependency pairs are usually easier than considering the original ones which are more general.

Definition 4.3.37 (instantiation processor)**.**

Let $\mathcal{P} = \mathcal{P}' \cup \{s \to t\}$. The *instantiation processor* is defined by

$$\phi_I((\mathcal{P}, \mathcal{R})) = \begin{cases} (\mathcal{P}' \cup \{s\mu \to t\mu \mid \mu = mgu(REN(CAP(v)), s), u \to v \in \mathcal{P}\}, \mathcal{R}) \\ (\mathcal{P}, \mathcal{R}) \end{cases} \quad \text{otherwise}$$

Example 4.3.38.

For example consider the DP problem $(\mathcal{P}, \mathcal{R})$, where $\mathcal{R}$ consists of eight rules

$$\begin{aligned} le(0, y) &\to true && (4.56)\\ div(x, s(y)) &\to if(le(s(y), x), x, s(y)) && (4.57)\\ le(s(x), 0) &\to false && (4.58)\\ if(true, x, y) &\to s(div(minus(x, y), y)) && (4.59)\\ le(s(x), s(y)) &\to le(x, y) && (4.60)\\ if(false, x, y) &\to 0 && (4.61)\\ minus(x, 0) &\to x && (4.62)\\ minus(s(x), s(y)) &\to minus(x, y) && (4.63) \end{aligned}$$

and $\mathcal{P}$ consists of two dependency pairs

$$\begin{aligned} DIV(s(x), s(y)) &\to IF(le(x, y), s(x), s(y)) && (4.64)\\ IF(true, x, y) &\to DIV(minus(x, y), y) && (4.65) \end{aligned}$$

In this example, the application of (4.64) may be used to precede the application of (4.65) on some DP chain. In this case, indeed only (4.64) can precede (4.65). We may apply the instantiation processor to instantiate the dependency pair (4.65). The right-hand side of (4.64) is $v = IF(le(x, y), s(x), s(y)))$ and the $REN(CAP(v))$ is $IF(z, s(x), s(y))$. This term is unifiable with the (variable renamed) left-hand side of (4.64): $IF(true, x', y')$, with mgu $\{z \mapsto true, x' \mapsto s(x), y' \mapsto s(y)\}$. Hence all the terms that will be reduced by (4.65) in some DP chain must be of the form $IF(true, s(x), s(y))$. Instead of considering the dependency pair (4.65) it is sufficient to consider its instance:

$$IF(true, s(x), s(y)) \to DIV(minus(s(x), s(y)), s(y)) \quad (4.66)$$

By using the instantiation processor, we can replace (4.65) with (4.66).

In practice, the same *safe* transformation as described in the narrowing processor can be applied to guarantee that the instantiation processes terminate.

Definition 4.3.39 (Safe instantiation transformations)**.**

Let $\mathcal{P} = (\mathcal{P}_1, \mathcal{R}_1)$ and $\mathcal{Q} = (\mathcal{P}_2, \mathcal{R}_2)$ be DP problems, and $\mathcal{Q}$ the result after some instantiation processes of $\mathcal{P}$. The instantiation processes are *safe* if

- $\mid \cup_{\mathcal{S} \in SCC(\mathcal{P})} \mathcal{S} \mid > \mid \cup_{\mathcal{S} \in SCC(\mathcal{Q})} \mathcal{S} \mid$, or
- $\{o(s \to t) \mid s \to t \in \cup_{\mathcal{S} \in SCC(\mathcal{P})} \mathcal{S}\} \supset \{o(s \to t) \mid s \to t \in \cup_{\mathcal{S} \in SCC(\mathcal{Q})} \mathcal{S}\}$

Both the narrowing and the instantiation processor, base their performance on transforming a dependency pair to some of its variants such that the finiteness problem become easier to be proved. The next two DP processors are no longer based on transformation, but they are directly derived from the termination proof method that is described on Chapter 2.

4.3.7 Match-bounds Processor

This processor is based on the termination proof method: *match-bounds*, that is already described in the previous chapter. The match-bounds method can be used to prove termination of the left-linear TRSs. This method states that if a left-linear TRS is either top-bounded, roof-bounded, or linear and match-bounded, then termination can be concluded. This idea can be adapted in the context of the DP framework. We define a *match-bounds processor* that can simplify a DP problem $(\mathcal{P}, \mathcal{R})$ if the combined TRS $\mathcal{P} \cup \mathcal{R}$ is also left-linear and either top-bounded, roof-bounded, or linear and match-bounded. If this is the case then it will transform the input DP problem to a finite DP problem $(\emptyset, \mathcal{R})$.

Definition 4.3.40 (match-bounds processor)**.**

The *match-bounds processor* is defined by

$$\phi_{MB}((\mathcal{P}, \mathcal{R})) = \begin{cases} (\emptyset, \mathcal{R}) & \text{if } \mathcal{P} \cup \mathcal{R} \text{ is left-linear and either top-bounded, roof-bounded, or linear and} \\ & \text{match-bounded} \\ (\mathcal{P}, \mathcal{R}) & \text{otherwise} \end{cases}$$

Theorem 4.3.41. *The match-bounds processor is sound and complete.*

This result is straightforwardly derived from Theorem 3.2.10.

Example 4.3.42.

For example, consider the DP problem $(\mathcal{P}, \mathcal{R})$ over the signature $\{a, f, g, h, F, H\}$, where $\mathcal{R}$ consists of two rules

$$f(g(x), y) \to g(h(x, y))$$
$$h(x, y) \to f(x, g(y))$$

and $\mathcal{P}$ consists of one dependency pair $F(g(x), y) \to G(h(x, y))$. Note that all the left and the right -hand sides of the rules in the combined TRS $\mathcal{P} \cup \mathcal{R}$ are linear. We can use the match-bounds processor to show finiteness of this DP problem by considering the enrichment $match$. The combined TRS $\mathcal{P} \cup \mathcal{R}$ is match-bounded by 2. Consider the TRS $match(\mathcal{P} \cup \mathcal{R})$ that consists of the rules

$$\begin{array}{rcl}
f_0(g_0(x), y) & \to & g_1(h_1(x, y)) \\
f_1(g_0(x), y) & \to & g_1(h_1(x, y)) \\
f_0(g_1(x), y) & \to & g_1(h_1(x, y)) \\
f_1(g_1(x), y) & \to & g_2(h_2(x, y)) \\
\vdots & & \\
h_0(x, y) & \to & f_1(x, g_1(y)) \\
h_1(x, y) & \to & f_2(x, g_2(y)) \\
\vdots & & \\
F_0(g_0(x), y) & \to & G_1(h_1(x, y)) \\
F_1(g_0(x), y) & \to & G_1(h_1(x, y)) \\
F_0(g_1(x), y) & \to & G_1(h_1(x, y)) \\
F_1(g_1(x), y) & \to & G_2(h_2(x, y)) \\
\vdots & &
\end{array}$$

All the terms that can be derived from $lift_0(\mathcal{T}(\mathcal{F}))$, by $match(\mathcal{P} \cup \mathcal{R})$ will have *height* at most 2. For example,

$$F_0(g_0(a_0), a_0) \to_{match(\mathcal{P} \cup \mathcal{R})} G_1(h_1(a_0, a_0)) \to_{match(\mathcal{P} \cup \mathcal{R})} G_1(f_2(a_0, g_2(a_0)))$$

This holds in general, and can be shown by giving a tree automata as described in the previous chapter. Hence by the match-bound processor we can transform the initial DP problem into $(\emptyset, \mathcal{R})$, and prove the original TRS $\mathcal{R}$ to be terminating.

In practice, the match-bounds processor either succeeds by proving that the combined left-linear TRS $\mathcal{P} \cup \mathcal{R}$ is top-bounded, roof-bounded, or both linear and match-bounded, or the processor fails when the match-, top-, or roof- boundedness of $\mathcal{P} \cup \mathcal{R}$ cannot be proved, and returns the initial DP problem. To show that the combined TRS $\mathcal{P} \cup \mathcal{R}$ is match-, top-, or roof- bounded, we construct iteratively a finite tree automaton [KM08]. If this procedure stops, then we can conclude that the DP problem is match-, top-, or roof- bounded. However the construction of this tree automaton does not terminate for TRSs that are not match-, top-, or roof- bounded. Hence, the situation where the match-bounds processor returns the initial DP problem typically does not happen. Because of this reason, the match-bounds processor in the Definition 4.3.40 is applicable only at the leaves of the DP Framework search tree, which means that it is used as the last option in the proving strategy.

The next DP processor is also derived from the general termination proof methods from Chapter 2. But unlike the match-bounds processor, this DP processor is not used as the last option in the proving strategy of the DP framework.

4.3.8 Semantic Labeling Processor

This processor is derived from the termination proof method: *semantic labeling*, that is also already mentioned in the previous chapter. The semantic labeling method tries to prove termination of a TRS by transforming it into a labeled TRS, and then prove termination of the original TRS in the labeled TRS. The idea of this method can also be adapted in the context of the DP framework. The basic idea of the semantic labeling processor is to simplify the finiteness problem of some DP problems by transforming them into labeled DP problems. Both the model and the quasi-model version of the semantic labeling (Definition 3.3.2) can be adapted as the semantic labeling processors. In the following, we give the adaptation for the quasi-model version.

Definition 4.3.43 (semantic labeling processor: quasi-model version)**.**
Let $\mathcal{A}$ be a weakly monotone algebra quasi-model of $\mathcal{R}$, and ℓ a labeling for $\mathcal{A}$. The *semantic labeling processor* is defined by

$$\phi_{SLQ}((\mathcal{P},\mathcal{R})) = (\mathcal{P}_{lab}, \mathcal{R}_{lab} \cup \mathcal{D}ec).$$

At first glance we may ask why the interpretation $\mathcal{A}$ is not needed to be a quasi-model for the dependency pairs $\mathcal{P}$. Since naturally, to label the combined TRS $\mathcal{P} \cup \mathcal{R}$, the algebra $\mathcal{A}$ should also be a quasi model for both $\mathcal{P}$ and $\mathcal{R}$. For this reason, first note that the interpretation $\mathcal{A}$ is needed to determine the labels of the function symbols. However, the labels of the function symbols depend on the interpretation of the arguments. On the other hand, the dependency pair symbols always occur at the root position. So no matter which interpretation is given to the dependency pair symbols, it will not affect any labels. Hence by choosing them to be the same constants we can get the quasi-model constraints for the dependency pairs for free.

There is also another refinement for this processor. We can incorporate the notion of *usable rules*, that is described in the section of the reduction pair processor. In [KM07], it shown that it is sufficient to require the algebra $\mathcal{A}$ to be a quasi-model for the usable rules, instead for all the rewrite rules. But this weaker restriction can only be applied if we add some additional property to the algebra $\mathcal{A}$. This property is, for every finite subset X in the domain of the interpretation A, there must exists a least upper bound $\sqcup X$ of X in A. We will call this interpretation $\sqcup$*-algebra* and this variant of labeling *predictive labeling* [HM06]. The refinement of the semantic labeling processor is given as follows:

Definition 4.3.44 (semantic labeling processor)**.**
Let $\mathcal{A}$ be a $\sqcup$-algebra quasi-model of $\mathcal{U}(\mathcal{P},\mathcal{R})$, and ℓ a labeling for $\mathcal{A}$. The *semantic labeling processor* is defined by

$$\phi_{SL}((\mathcal{P},\mathcal{R})) = (\mathcal{P}_{lab}, \mathcal{R}_{lab} \cup \mathcal{D}ec).$$

Theorem 4.3.45. *The semantic labeling processor is sound and complete.*

The complete proof of this theorem is described in [KM07].

Example 4.3.46 ([KM07])**.**

We will give an example that illustrates the application of the semantic labeling processor (Definition 4.3.44). Consider the TRS $\mathcal{R}$ of addition and Ackermann function that consists of seven rules

$$\begin{aligned}
plus(0, y) &\to y \\
plus(s(0), y) &\to s(y) \\
plus(s(s(x)), y) &\to s(plus(x, s(y))) \\
plus(x, s(s(y))) &\to s(plus(s(x), y)) \\
ack(0, y) &\to s(y) \\
ack(s(x), 0) &\to ack(x, s(0)) \\
ack(s(x), s(y)) &\to ack(x, plus(y, ack(s(x), y)))
\end{aligned}$$

The dependency pairs $\mathcal{DP}(\mathcal{R})$ are

$$\begin{aligned}
ACK(s(x), 0) &\to ACK(x, s(0)) && (4.67)\\
PLUS(s(s(x)), y) &\to PLUS(x, s(y)) && (4.68)\\
PLUS(x, s(s(y))) &\to PLUS(s(x), y) && (4.69)\\
ACK(s(x), s(y)) &\to ACK(x, plus(y, ack(s(x), y))) && (4.70)\\
ACK(s(x), s(y)) &\to PLUS(y, ack(s(x), y)) && (4.71)\\
ACK(s(x), s(y)) &\to ACK(s(x), y) && (4.72)
\end{aligned}$$

First we can decompose the initial DP problem into two DP problems by using the dependency graph processor. The estimated dependency graph by unification is depicted in Figure 4.7.

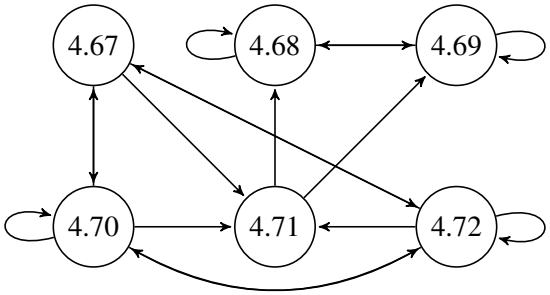

Figure 4.7: Dependency graph

There are two SCCs, $\mathcal{P}_1 = \{(4.67), (4.70), (4.72)\}$ and $\mathcal{P}_2 = \{(4.68), (4.69)\}$. The first DP problem ($\mathcal{P}_1$, $\mathcal{R}$) can be solved by two consecutive applications of the subterm criterion processor. The first one with the argument projection to the first argument of ACK will delete

(4.67) and (4.70). Then the second projection is done to the second argument of ACK, to delete (4.72). Hence we obtain a finite DP problem from the first DP problem.

We will show that we can solve the second DP problem: ($\mathcal{P}_2$, $\mathcal{R}$) = ($\{(4.68), (4.69)\}$, $\mathcal{R}$) by first applying the semantic labeling processor, and then the reduction pair processor. In this case we can use a polynomial interpretation $\mathcal{P}$ which interprets the function symbol s as $s_{\mathcal{P}}(x) = x + 1$. Note that the DP problem ($\{(4.68), (4.69)\}$, $\mathcal{R}$) has no usable rules. Hence this interpretation is already sufficient to apply the semantic labeling processor from Definition 4.3.44. Moreover, in this interpretation we define a labeling ℓ which labels the function symbol $PLUS$ with $\ell_{PLUS}(x, y) = x + y$. Hence we obtain $\mathcal{D}ec = \{PLUS_i(x, y) \to PLUS_j(x, y) \mid i > j \geq 0\}$, and the labeled dependency pairs from (4.68) and (4.69) are

$$\begin{aligned} PLUS_{i+j+2}(s(s(x)), y) &\to PLUS_{i+j+1}(x, s(y)) \\ PLUS_{i+j+2}(x, s(s(y))) &\to PLUS_{i+j+1}(s(x), y) \end{aligned}$$

for all $i, j \geq 0$. Here the labeled rewrite rules remain the same, $\mathcal{R}_{lab} = \mathcal{R}$.

Now we will apply the reduction pair processor to the DP problem $(\{(4.68)_{lab}, (4.69)_{lab}\}, \mathcal{R} \cup \mathcal{D}ec)$. First note again that there is no usable rules in this DP problem. Hence we do not need to search for a weak order to orient any rewrite rules. We only need to find a strict order that can orient the dependency pairs that are infinitely many. Consider the argument filtering $\pi(PLUS_i)$ = [ ], and an LPO induced by the precedence $PLUS_i > PLUS_j$ for every $i > j$. Then all labeled dependency pairs $(4.68)_{lab}$, $(4.69)_{lab}$ and $\mathcal{D}ec$ are strictly decreasing with respect to this order. By the definition of the reduction pair processor, we can delete all the dependency pairs, and obtain a finite DP problem $(\emptyset, \mathcal{R} \cup \mathcal{D}ec)$. Since we obtain a finite DP problem, the initial DP problem is also finite. Hence the original TRS $\mathcal{R}$ has successfully been proved terminating. The termination proof in the DP framework is depicted in Figure 4.8.

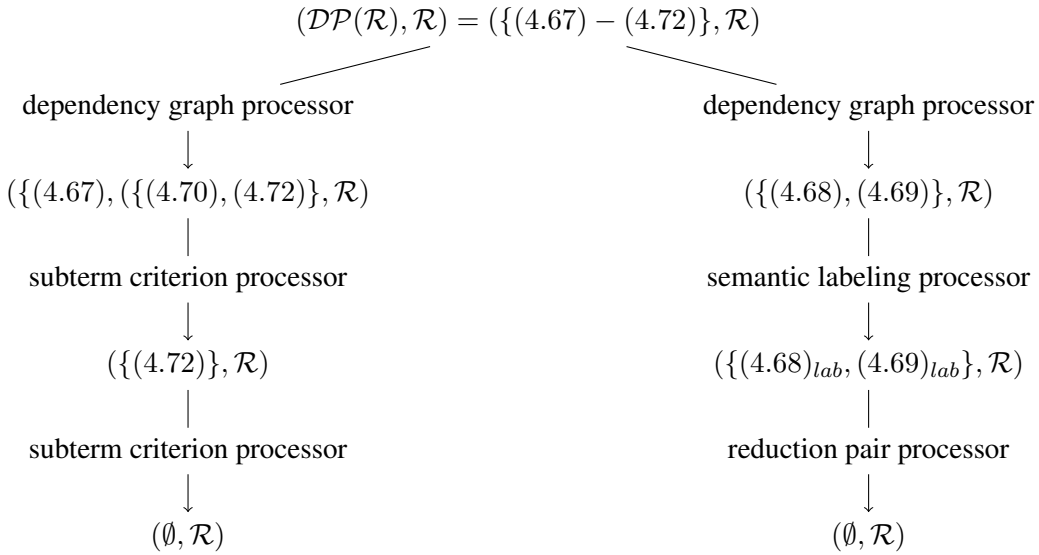

Figure 4.8: The dependency pair framework

In [KM07], the issue of the practical implementation of the semantic labeling processor is discussed. Similar to the original method that is already mentioned in the previous chapter, the set of the natural numbers is used as the domain of the interpretation, and polynomial interpretations is used to interpret the function symbols. The application of the semantic labeling processor is also restricted only if in the labeled DP problems, at least one rule can be deleted by the reduction pair processor induced by LPO. All these restrictions are translated into a propositional logic formula and in [KM07] a SAT solver is called to return the possible interpretations, and the possible precedences of the LPOs.

In this chapter we have briefly described the termination proof techniques used in the DP framework. At the beginning of this chapter, we have described the basic idea of the dependency pairs and the motivation behind the termination criterion underlying the DP framework. Then, we have presented the formulation of the DP framework, and the mechanism how to obtain termination proofs by applying the DP processors. As the main part of this chapter we gave a comprehensive survey of various types of DP processors. Along the summary of many processors, we have shown some integrated examples to illustrate proving termination with the DP framework. We have seen that from one single termination problem, we can decompose it to some subproblems, and solve them separately and gradually by using different termination techniques (DP processors). This idea increases significantly the flexibility, modularity, and the power of proving termination. We have shown that some TRSs that are difficult or cannot be solved with the direct methods, now can be solved by using the DP framework.

CHAPTER 5

Conclusion

We have summarized the method of proving termination with the DP framework. We have started this by giving a short survey of some general methods, that can also be applied in the the DP framework. We covered three different types of the general termination proof methods according to the survey paper [Zan00]: semantical, syntactical, and transformational methods. We discussed some general methods like polynomial interpretations, matrix interpretations, simplification orders, match-bounds, and semantic labeling. For each method we have presented the basic ideas, theoretical foundations, and some examples to illustrate its application.

Then as the main part of this thesis, we have summarized comprehensively the method of the DP framework. We have presented the main ideas, the underlying theory, and the formulation of the DP framework. We gave a comprehensive survey of DP processors, since basically the termination proof in the DP framework is nothing more but repeated applications of DP processors. We cover the most important types of DP processors: the DP processor that can decompose termination problems (dependency graph processor), the DP processor based on searching for orders (reduction pair processor, rule removal processor), the DP processors that can accommodate applications of other DP processors (narrowing processor, instantiation processor), and the DP processors that are directly derived from other general methods (match-bounds processor, semantic labeling processor). For each DP processor, the main idea, its theoretical foundation, examples, and some possible refinements are given.

From this concise summary, we conclude two characteristics of proving termination with the DP framework. The first one is: all the termination techniques used in the DP framework have to be uniformly formulated as DP processors. Moreover, the DP processors are needed to be sound (complete resp.), in order to be able to prove termination (non-termination resp.) correctly. The second characteristic that we can derive is: the DP processors indeed may adapt the principle of other general termination techniques. It has been shown that there are some DP processors that are straightforwardly derived from the general methods, like the match-bounds and the semantic labeling processor. There are also some DP processors that are not straightforwardly derived, but still use the principle of other general methods. For example, the reduction pair processor and the

rule removal processor adapt the principles of the reduction orders, and may use the techniques from polynomial interpretations, RPOs, and KBOs to search for some suitable reduction pairs.

In summary, we have presented a structured and detailed survey on proving termination of rewriting with the dependency pair framework, which is very flexible, modular and powerful. It would also be interesting to investigate in detail strategies used in the DP framework for controlling the use of DP processors. Some further possible works can be related to collecting the strategies in using the DP processors, which we haven't covered in this summary.

Bibliography

[AG00] Thomas Arts and Jürgen Giesl. Termination of rerm rewriting using dependency pairs. *Theoretical Computer Science*, 236:133-178, 2000.

[Dav73] Martin Davis. Hilbert's tenth problem is unsolvable. *The Amer. Math. Monthly*, 80:233-269, 1973.

[EWZ06] Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. In *Proc. of the 3rd International Joint Conference on Automated Reasoning (IJCAR '06), volume 4130 of Lecture Notes in Computer Science*, pages 574-588, Springer, 2006.

[FGMSTZ07] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. SAT solving for termination analysis with polynomial interpretations. In *Proc. of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT '07), volume 4501 of Lecture Notes in Computer Science*, pages 340-354, 2007.

[GHW03] Alfons Geser, Dieter Hofbauer, and Johannes Waldmann. Match-bounded string rewriting systems. In *Proc. of the 28th International Symposium on Mathematical Foundations of Computer Science (MFCS '03), volume 2747 of Lecture Notes in Computer Science*, pages 449-459, 2003.

[GHWZ05] Alfons Geser, Dieter Hofbauer, Johannes Waldmann, and Hans Zantema. On tree automata that certify termination of left-linear term rewriting systems. In *Proc. of the 16th International Conference on Rewriting Techniques and Applications (RTA '05), volume 3467 of Lecture Notes in Artificial Intelligence*, pages 353-267, 2005.

[GSST06] Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp, and Réne Thiemann. Automated termination analysis for Haskell: From term rewriting to programming languages. In *Proc. of the 17th International Conference on Rewriting Techniques and Applications (RTA '06), volume 4098 of Lecture Notes in Computer Science*, pages 297-312, 2006.

[GTS05] Jürgen Giesl, Réne Thiemann, and Peter Schneider-Kamp. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In Franz Baader and Andrei Voronkov, editors, *Proc. of 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, (LPAR '04)*, *Lecture Notes in Computer Science 3452*, pages 301-331, Springer-Verlag, 2004.

[GTSF03] Jürgen Giesl, Réne Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing dependency pairs. *Technical report AIB-2003-08, RWTH Aachen, Germany*. Available from http://aib.informatik.rwth-aachen.de, 2003.

[GTSF06] Jürgen Giesl, Réne Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155-203, 2006.

[HJ98] Hoon Hong and Dalibor Jakuš. Testing Positiveness of Polynomials. Journal of Automated Reasoning, volume 21(1), pages 23-38, 1998.

[HL78] Gérard Huet and Dallas Lankford. On the uniform halting problem for term rewriting systems. Technical Report 283, IRIA, 1978.

[HM04] Nao Hirokawa and Aart Middeldorp. Dependency Pairs Revisited. In Vincent van Oostrom, editor, *Proc. 15th International Conference on Rewriting Techniques and Applications (RTA '04). Lecture Notes in Computer Science 3091*, pages 249-268, ACM Press, 2004.

[HM06] Nao Hirokawa and Aart Middeldorp. Predictive labeling. In *Proc. of the 17th International Conference on Rewriting Techniques and Applications (RTA '06)*, *Lecture Notes in Computer Science 4098*, pages 313-327. 2006.

[HW06] Dieter Hofbauer and Johannes Waldmann. Proving termination with matrix interpretations. In Frank Pfenning, editor, *Proc. of the 17th Conference on Rewriting Techniquesand Applications (RTA'06)*, *Lecture Notes in Computer Science*. Springer, 2006.

[KM07] Adam Koprowski and Aart Middeldorp. Predictive labeling with dependency pairs using SAT. In *Proc. of the 21st Conference on Automated Deduction (CADE '07), volume 4603 of Lecture Notes in Artificial Intelligence*, pages 410-425, 2007.

[KM08] Martin Korp and Aart Middeldorp. Match-bounds with dependency pairs for proving termination of rewrite systems. In *Proc. of the 2nd International Conference on Language and Automata Theory and Applications (LATA), volume 5196 of Lecture Notes in Computer Science*, pages 321-332, 2008.

[KW08] Adam Koprowski and Johannes Waldmann. Arctic termination . . . below zero. In Andrei Voronkov editors, *Proc. of the 19th International Conference on Rewriting Techniques and Applications (RTA'08)*, *Lecture Notes in Artificial Intelligence 5117*, pages 202-216. Springer-Heidelberg, 2008.

[KZ06] Adam Koprowski and Hans Zantema. Automation of recursive path ordering for infinite labelled rewrite systems. In *Proc. of the 3rd International Joint Conference on Automated Reasoning (IJCAR '06), volume 4130 of Lecture Notes in Artificial Intelligence*, pages 332-346, 2006.

[Mid01] Aart Middeldorp. Approximating Dependency Graphs using Tree Automata techniques. In Fabio Massacci, Rajeev Goré, editors, *Proc. 1st Int. Joint Conference Automated Reasoning*, *Lecture Notes in Artificial Intelligence 2083*, pages 593-610, Springer-Verlag, 2001.

[MK09] Aart Middeldorp and Martin Korp. *Match-bounds revisited*. Information and Computation, Volume 207:11, pages 1259-1283, 2009.

[New42] Maxwell Newman. On theories with a combinatorial definition of "equivalence". Annals of Mathematics, volume 43(2), pages 223-243, 1942.

[Sal04] Salvador Lucas. Polynomials over the reals in proof of termination. *In Proc. 7th International Workshop on Termination, Technical Report AIB-2004-07, RWTH Aachen*, pages 39-42, 2004.

[SGST06] Peter Schneider-Kamp, Jürgen Giesl, Alexander Serebrenik, and Réne Thiemann. Automated termination analysis for logic programs by term rewriting. In *Proc. of the 16th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR '06), volume 4407 of Lecture Notes in Computer Science*, pages 177-193, 2006.

[Zan95] Hans Zantema. Termination of Term Rewriting by Semantic Labeling. *Fundamenta Informaticae*, 24:89-105, 1995.

[Zan00] Hans Zantema. Termination of Term Rewriting. *Technical Report UU-CS-2000, Department of Information and Computing Sciences, Utrecht University*, 2000.