Universidad Politécnica de Valencia

Departamento de Sistemas Informáticos y Computación

Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información

Master Thesis

Termination of Narrowing with Dependency Pairs

Candidate:

José Iborra

Supervisor:

María Alpuente, Santiago Escobar

Academic Year – December 2008 –

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia
España

Contents

Abstract

In this work, we extend the dependency pair approach for automated proofs of termination in order to prove the termination of narrowing. Our extension of the dependency pair approach generalizes the standard notion of dependency pairs by taking specifically into account the dependencies between the left-hand side of a rewrite rule and its own argument subterms. We demonstrate that the new *narrowing dependency pairs* exactly capture the narrowing termination behavior and provide an effective termination criterion which we prove to be sound and complete. Finally, we discuss how the problem of analyzing narrowing chains can be recast as a standard analysis problem for traditional (rewriting) chains, so that the proposed technique can be effectively mechanized by reusing the standard DP infrastructure.

1 Introduction

1.1 Narrowing

Narrowing [Fay, 1979] is a generalization of term rewriting that allows free variables in terms (as in logic programming) and replaces pattern matching by syntactic unification so that it subsumes both rewriting and SLD-resolution [Hanus, 1994]. Narrowing has many important applications including:

- execution of functional–logic programming languages [Dershowitz, 1995; Hanus, 1994; Meseguer, 1992],
- verification of security policies [Kirchner, Kirchner and de Oliveira, 2008] and cryptographic protocols [Escobar, Meadows and Meseguer, 2006],
- equational unification [Hullot, 1980*a*],
- equational constraint solving [Alpuente et al., 1993; Alpuente, Falaschi and Levi, 1995],
- symbolic reachability [Meseguer and Thati, 2007],
- automated proofs of termination for term rewriting systems [Arts and Zantema, 1996; Giesl et al., 2006],
- type checking [Sheard, 2006],
- and model checking [Escobar and Meseguer, 2007].

Termination of narrowing itself is of great interest to these applications. Without it, many of these applications are simply not possible or their usefulness is seriously affected. For instance, for all the applications related with reasoning, such as unification, verification and reachability, the lack of a termination proof means that narrowing only provides a semidecision procedure, whereas with a warranty of termination one gets a full decision procedure.

Termination of narrowing is a more restrictive property than termination of rewriting or termination of pure logic programs due to the high degree of nondeterminism caused by the interaction of rule selection, redex selection, and unification. In recent works [Alpuente, Escobar and Iborra, 2008*a*, *b*], we identified some non–trivial classes of TRSs where narrowing terminates. The results in [Alpuente, Escobar and Iborra, 2008*b*] generalize previously known criteria for termination of narrowing, which were essentially restricted before to either confluent term rewriting systems (TRSs) [Hullot, 1980*a*] or to left–flat TRSs (i.e., each argument of the left–hand side of a rewrite rule is either a variable or a ground term) that are compatible with a termination ordering [Christian, 1992], among other applicability conditions. Roughly speaking, we proved in [Alpuente, Escobar and Iborra, 2008*b*] that confluence is a superfluous requirement. We also weakened the left–flatness condition required in [Christian, 1992] to the requirement that every non-ground, strict subterm of the left–hand side (lhs) of every rewrite rule must be a *rigid normal form,* i.e., unnarrowable. Finally, in [Alpuente, Escobar and Iborra, 2008*a*] we proved modular termination of a restriction of narrowing, called basic narrowing [Hullot, 1980*a*], in several hierarchical combinations of TRSs, which provides new algorithmic criteria to prove termination of narrowing via termination of basic narrowing (cf. [Alpuente, Escobar and Iborra, 2008*b*]).

1.2 Dependency Pairs

In recent years, the dependency pair (DP) method for automating the termination proofs of term rewriting has achieved tremendous success, as witnessed by the large number of publications and tools since its introduction in [Arts and Giesl, 2000] and subsequent reformulation in [Giesl, Thiemann and Schneider-Kamp, 2005] (see [Giesl et al., 2006; Hirokawa and Middeldorp, 2004] for extensive references thereof). In [Nguyen et al., 2008], the notions of dependency pairs and dependency graphs, which were originally developed for term rewriting, were adapted to the logic programming domain, leading to automated termination analyses that are directly applicable to any definite logic program.

Two different adaptations of the DP technique for narrowing have been proposed recently. In [Nishida and Miura, 2006; Nishida, Sakai and Sakabe, 2003], the original dependency pair technique of [Arts and Giesl, 2000] was adapted to the termination of narrowing, whereas [Nishida and Vidal, 2008] adapts the logic programming dependency pair approach of [Nguyen et al., 2008] instead, to prove termination of narrowing w.r.t. a given set of queries that are supplemented with *call modes*. Unfortunately, these two methods apply only to two particular classes of TRSs: right–linear TRSs (i.e., no repeated variables occur in the right–hand sides of the rules) or constructor systems (the arguments of the lhs's of the rules are constructor –i.e., data– terms). These two classes are overly restrictive for many practical uses of narrowing, such as the applications mentioned above. In this work, we are able to relax these restrictions and provide a method which is applicable to any class of TRSs.

Example 1 *Consider our running example, which is the non–right–linear, non–constructor–based, non–confluent TRS adapted from [Kirchner, Kirchner and de Oliveira, 2008], shown*[1] *in Figure 1.1. This TRS models a security (filtering) and routing policy that allows packets coming from external networks to be analyzed. We do not describe the intended meaning of each symbol since it is not relevant for this work, but note the kind of expressivity that is assumed in the domain of rule–based policy specification, which does not fit in the right–linear restriction or the constructor discipline. Narrowing is terminating for this TRS, but it cannot be proved by using any of the existing methods [Alpuente, Escobar and Iborra, 2008*a, b*; Nishida and Miura, 2006; Nishida, Sakai and Sakabe, 2003; Nishida and Vidal, 2008]. In this paper, we*

[1] In this paper, variables are written in italic font and function symbols are in typewriter font.

$$
\begin{array}{lcl}
\mathtt{filter}(\mathtt{pckt}(src, dst, \mathtt{established})) & \to & \mathtt{accept} \\
\mathtt{filter}(\mathtt{pckt}(\mathtt{eth0}, dst, \mathtt{new})) & \to & \mathtt{accept} \\
\mathtt{filter}(\mathtt{pckt}(\mathtt{194.179.1.}x\mathtt{:}port, dst, \mathtt{new})) & \to & \mathtt{filter}(\mathtt{pckt}(\mathtt{secure}, dst, \mathtt{new})) \\
\mathtt{filter}(\mathtt{pckt}(\mathtt{158.42.}x.y\mathtt{:}port, dst, \mathtt{new})) & \to & \mathtt{filter}(\mathtt{pckt}(\mathtt{secure}, dst, \mathtt{new})) \\
\mathtt{filter}(\mathtt{pckt}(\mathtt{secure}, dst\mathtt{:80}, \mathtt{new})) & \to & \mathtt{accept} \\
\mathtt{filter}(\mathtt{pckt}(\mathtt{secure}, dst\mathtt{:other}, \mathtt{new})) & \to & \mathtt{drop} \\
\mathtt{filter}(\mathtt{pckt}(\mathtt{ppp0}, dst, \mathtt{new})) & \to & \mathtt{drop} \\
\mathtt{filter}(\mathtt{pckt}(\mathtt{123.123.1.1:}port, dst, \mathtt{new})) & \to & \mathtt{accept} \\
\mathtt{pckt}(\mathtt{10.1.1.1:}port, \mathtt{ppp0}, s) & \to & \mathtt{pckt}(\mathtt{123.23.1.1:}port, \mathtt{ppp0}, s) \\
\mathtt{pckt}(\mathtt{10.1.1.2:}port, \mathtt{ppp0}, s) & \to & \mathtt{pckt}(\mathtt{123.23.1.1:}port, \mathtt{ppp0}, s) \\
\multicolumn{3}{l}{\mathtt{pckt}(src, \mathtt{123.123.1.1:}port, \mathtt{new}) \to \mathtt{natroute}(\mathtt{pckt}(src, \mathtt{10.1.1.1:}port, \mathtt{established}),} \\
 & & \mathtt{pckt}(src, \mathtt{10.1.1.2:}port, \mathtt{established})) \\
\mathtt{natroute}(a, b) & \to & a \\
\mathtt{natroute}(a, b) & \to & b
\end{array}
$$

Figure 1.1: The *FullPolicy* TRS

develop techniques that allow us to prove it automatically.

Plan of the thesis

The main contributions of this thesis are as follows:

- We present a new method for proving the termination of narrowing that is based on a suitable extension of the DP technique to narrowing that is applicable to any class of TRSs. Our method generalizes the standard notion of dependency pairs to narrowing by taking the dependencies between the lhs of a rewrite rule and its own argument subterms specifically into account.

- We demonstrate that the new *narrowing dependency pairs* exactly capture the termination of narrowing behavior. We provide a termination criterion based on *narrowing chains* which we show to be sound and complete.

- This allows us to develop a technique that is more general in all cases and, for general calls (i.e., without considering call modes) strictly subsumes the DP methods for proving termination of narrowing of [Nishida and Miura, 2006; Nishida, Sakai and Sakabe, 2003; Nishida and Vidal, 2008], as well as all previous (decidable) termination of narrowing criteria [Alpuente, Escobar and Iborra, 2008*a*, *b*; Christian, 1992; Hullot, 1980*a*].

- We have implemented a tool for proving the termination of narrowing automatically that is based on our technique, and we made it publicly available.

After recalling some preliminaries in Section 2, in Section 3.1 we discuss the problem of *echoing*, which we identify as being ultimately responsible for the non–termination of narrowing. In Section 3.2, we develop the notion of narrowing dependency pairs and provide a sound and complete criterion for the termination of narrowing that is based on analyzing narrowing chains. In Section 3.3, we discuss the effective automation of our method, which mainly consists of two steps: DP extraction and argument filtering transformation. In Section 4 we discuss in detail the implementation of an automatic tool based on the method developed in this thesis. Section 5 concludes with an analysis of the state of the art, and the possible directions for future work in this line.

2 Preliminaries

2.1 Narrowing

We now briefly recall the essential notions and terminology of term rewriting. For missing notions and definitions on equations, orderings and rewriting, we refer to [TeReSe, 2003].

Terms, variables and positions. $\mathcal{V}$ denotes a countably infinite set of variables, and Σ denotes a set of function symbols, or signature, each of which has a fixed associated arity. Terms are viewed as labelled trees in the usual way, where $\mathcal{T}(\Sigma, \mathcal{V})$ and $\mathcal{T}(\Sigma)$ denote the non-ground term algebra and the ground algebra built on $\Sigma \cup \mathcal{V}$ and Σ, respectively. Positions are defined as sequences of natural numbers used to address subterms of a term, with ϵ as the root (or top) position (i.e., the empty sequence). Concatenation of positions p and q is denoted by $p.q$, and $p < q$ is the usual prefix ordering. The root symbol of a term is denoted by root(t). Given $S \subseteq \Sigma \cup \mathcal{V}$, $\mathcal{P}os_S(t)$ denotes the set of positions of a term t that are rooted by function symbols or variables in S. $\mathcal{P}os_{\{f\}}(t)$ with $f \in \Sigma \cup \mathcal{V}$ will be simply denoted by $\mathcal{P}os_f(t)$, and $\mathcal{P}os_{\Sigma \cup \mathcal{V}}(t)$ will be simply denoted by $\mathcal{P}os(t)$. $t|_p$ is the subterm at the position p of t. $t[s]_p$ is the term t with the subterm at the position p replaced with term s. By $Var(s)$, we denote the set of variables occurring in the syntactic object s. By $\bar{x}$, we denote a tuple of pairwise distinct variables. A *fresh* variable is a variable that appears nowhere else. A *linear* term is one where every variable occurs only once.

Substitutions, unifiers. A *substitution* σ is a mapping from the set of variables $\mathcal{V}$ into the set of terms $\mathcal{T}(\Sigma, \mathcal{V})$ with a (possibly infinite) domain $D(\sigma)$, and image $I(\sigma)$. A substitution is represented as $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ for variables $x_1, \ldots, x_n$ and terms $t_1, \ldots, t_n$. The application of a substitution θ to term t is denoted by $t\theta$, using postfix notation. Composition of substitutions is denoted by juxtaposition, i.e., the substitution $\sigma\theta$ denotes $(\theta \circ \sigma)$. We write $\theta_{\restriction Var(s)}$ to denote the restriction of the substitution θ to the set of variables in s; by abuse of notation, we often simply write $\theta_{\restriction s}$. Given a term t, $\theta = \nu\ [t]$ iff $\theta_{\restriction Var(t)} = \nu_{\restriction Var(t)}$, that is, $\forall x \in Var(t)$, $x\theta = x\nu$. A substitution θ is more general than σ, denoted by $\theta \leq \sigma$, if there is a substitution γ such that $\theta\gamma = \sigma$. A *unifier* of terms s and t is a substitution ϑ such that $s\vartheta = t\vartheta$. The *most general unifier* of terms s and t, denoted by $mgu(s, t)$, is a unifier θ such that for any other unifier θ', $\theta \leq \theta'$.

Rewriting. A *term rewriting system* (TRS) $\mathcal{R}$ is a pair (Σ, R), where R is a finite set of rewrite rules of the form $l \to r$ such that $l, r \in \mathcal{T}(\Sigma, \mathcal{V})$, $l \notin \mathcal{V}$, and $Var(r) \subseteq Var(l)$. For TRS $\mathcal{R}$, $l \to r \ll \mathcal{R}$ denotes that $l \to r$ is a new variant of a rule in $\mathcal{R}$ such that $l \to r$ contains only *fresh* variables, i.e., contains no variable previously met during any computation (standardized apart). We will often write just $\mathcal{R}$ or (Σ, R) instead of $\mathcal{R} = (\Sigma, R)$. A TRS $\mathcal{R}$ is called *left–linear* (respectively *right–linear*) if, for every $l \to r \in \mathcal{R}$, l (respectively r) is a linear term. Given a TRS $\mathcal{R} = (\Sigma, R)$, the signature Σ is often partitioned into two disjoint sets $\Sigma = \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{D} = \{f \mid f(t_1, \ldots, t_n) \to r \in R\}$ and $\mathcal{C} = \Sigma \setminus \mathcal{D}$. Symbols in $\mathcal{C}$ are called *constructors*, and symbols in $\mathcal{D}$ are called *defined functions.* The elements of $\mathcal{T}(\mathcal{C}, \mathcal{V})$ are called *constructor terms.* We often introduce a TRS as $\mathcal{R}(\mathcal{D}, \mathcal{C}, R)$. We let $Def(\mathcal{R})$ denote the set of defined symbols in $\mathcal{R}$. A constructor system is a TRS whose lhs's are terms of the form $f(c_1, \ldots, c_k)$ where $f \in \mathcal{D}$ and $c_1, \ldots, c_k$ are constructor terms. A term whose root symbol is a defined function is called *root-defined.*

A rewrite step is the application of a rewrite rule to an expression. A term $s \in \mathcal{T}(\Sigma, \mathcal{V})$ *rewrites* to a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, denoted by $s \xrightarrow{p}_{\mathcal{R}} t$, if there exist $p \in \mathcal{P}os_{\Sigma}(s)$, $l \to r \in \mathcal{R}$, and substitution σ such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$. When no confusion can arise, we omit the subscript in $\to_{\mathcal{R}}$. We also omit the reduced position p when it is not relevant. A term s is a *normal form* w.r.t. the relation $\rightarrow_{\mathcal{R}}$ (or simply a normal form), if there is no term t such that $s \rightarrow_{\mathcal{R}} t$. A term is a reducible expression or *redex* if it is an instance of the left hand side of a rule in $\mathcal{R}$. A term s is a *head normal form* if there are no terms t, t' s.t. $s \rightarrow^*_{\mathcal{R}} t' \overset{\epsilon}{\rightarrow}_{\mathcal{R}} t$. A term t is said to be terminating w.r.t. R if there is no infinite reduction sequence $t \rightarrow_{\mathcal{R}} t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} \ldots$. A TRS $\mathcal{R}$ is *($\rightarrow$)-terminating* (also called strongly normalizing or noetherian) if every term is terminating w.r.t. R. A TRS $\mathcal{R}$ is *confluent* if, whenever $t \rightarrow^*_{\mathcal{R}} s_1$ and $t \rightarrow^*_{\mathcal{R}} s_2$, there exists a term w s.t. $s_1 \rightarrow^*_{\mathcal{R}} w$ and $s_2 \rightarrow^*_{\mathcal{R}} w$.

Narrowing. A term $s \in \mathcal{T}(\Sigma, \mathcal{V})$ narrows to a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, denoted by $s \overset{p}{\rightsquigarrow}_{\theta,\mathcal{R}} t$, if there exist $p \in \mathcal{P}os_{\Sigma}(s)$, $l \rightarrow r \ll \mathcal{R}$, and substitution θ such that $\theta = mgu(s|_p, l)$ and $t = (s[r]_p)\theta$. We use $\overset{>\epsilon}{\rightarrow}_{\mathcal{R}}$ (resp. $\overset{>\epsilon}{\rightsquigarrow}_{\theta,\mathcal{R}}$) to denote steps in which the selected redex (resp. *narrex*, i.e. narrowable expression) is below the root.

Example 2 *Consider the following term rewriting system (TRS) defining the addition* `add` *on natural numbers built from* `0` *and* `s`*:*

$$\mathtt{add}(\mathtt{0}, y) \rightarrow y \tag{R1}$$

$$\mathtt{add}(\mathtt{s}(x), y) \rightarrow \mathtt{s}(\mathtt{add}(x, y)) \tag{R2}$$

There are infinitely many narrowing derivations issuing from the input expression $\mathtt{add}(w, \mathtt{s}(\mathtt{0}))$ *(at each step, the narrowing relation* $\rightsquigarrow$ *is labelled with the applied substitution and rule*[1]*, and the reduced subterm is underlined):*

$$\begin{array}{ll}
\underline{\mathtt{add}(w, \mathtt{s}(\mathtt{0}))} & \rightsquigarrow_{\{w\mapsto \mathtt{0}\},(\mathrm{R1})} \mathtt{s}(\mathtt{0}) \\
\underline{\mathtt{add}(w, \mathtt{s}(\mathtt{0}))} & \rightsquigarrow_{\{w\mapsto \mathtt{s}(x)\},(\mathrm{R2})} \mathtt{s}(\underline{\mathtt{add}(x, \mathtt{s}(\mathtt{0}))}) \rightsquigarrow_{\{x\mapsto \mathtt{0}\},(\mathrm{R1})} \mathtt{s}(\mathtt{s}(\mathtt{0})) \\
\underline{\mathtt{add}(w, \mathtt{s}(\mathtt{0}))} & \rightsquigarrow_{\{w\mapsto \mathtt{s}(x)\},(\mathrm{R2})} \mathtt{s}(\underline{\mathtt{add}(x, \mathtt{s}(\mathtt{0}))}) \rightsquigarrow_{\{x\mapsto \mathtt{s}(x')\},(\mathrm{R2})} \mathtt{s}(\mathtt{s}(\underline{\mathtt{add}(x', \mathtt{s}(\mathtt{0}))})) \\
 & \rightsquigarrow_{\{x\mapsto \mathtt{0}\},(\mathrm{R1})} \mathtt{s}(\mathtt{s}(\mathtt{s}(\mathtt{0}))) \\
\vdots &
\end{array}$$

The following infinite narrowing derivation resulting from applying rule (R2) infinitely many times can also be proved

[1]Substitutions are restricted to the input variables.

$$\underline{\mathtt{add}(w, \mathtt{s}(\mathtt{0}))} \leadsto_{\{w \mapsto \mathtt{s}(x)\},(\mathrm{R2})} \mathtt{s}(\underline{\mathtt{add}(x, \mathtt{s}(\mathtt{0}))}) \leadsto_{\{x \mapsto \mathtt{s}(x')\},(\mathrm{R2})} \mathtt{s}(\mathtt{s}(\underline{\mathtt{add}(x', \mathtt{s}(\mathtt{0}))})) \cdots$$

Due to nontermination, narrowing behaves as a semi-decision procedure for the problem of equational unification in a wide variety of equational theories. For instance, in the equational theory defined by the above rules (R1) and (R2), narrowing allows us to prove that the formula $\exists w \exists z$ s.t. $\mathtt{add}(w, \mathtt{s}(\mathtt{0})) = \mathtt{s}(\mathtt{s}(z))$ holds by computing the solution $\{w \mapsto \mathtt{s}(\mathtt{0}),\ z \mapsto 0\}$, whereas it cannot prove that the formula $\exists w$ s.t. $\mathtt{add}(w, \mathtt{s}(\mathtt{0})) = \mathtt{0}$ does not hold.

Under appropriate conditions, narrowing is complete as an equational unification algorithm as well as a procedure to solve *reachability* problems; that is, it is able to find "more general" solutions σ for the variables of terms s and t, such that $s\sigma$ rewrites to $t\sigma$ in $\mathcal{R}$ in a number of steps. For instance, narrowing computes the solution $\{w \mapsto \mathtt{s}(z)\}$ for the reachability problem $\exists w \exists z$ s.t. $\mathtt{add}(\mathtt{0}, w) \rightarrow^* \mathtt{s}(z)$.

2.2 The Dependency Pairs method

The dependency pair technique [Arts and Giesl, 2000] is one of the most powerful methods for automated termination analyses. The technique has experimented several steps of evolution since it was originally introduced, and in its current status it is commonly known as the DP framework, described extensively in [Thiemann, 2007]. In order to make this thesis self-contained, in this section we give a very brief introduction to the dependency pairs method.

Intuitively, the technique focuses on the dependency relations between defined function symbols, paying particular attention to strongly connected components within a graph of functional dependencies, in order to produce automated termination proofs. The dependency graph is extracted by considering the dependencies between the lhs's of a rewrite rule and all proper subterms of the rhs of the rule. More precisely, if a term $\mathtt{f}(t_1 \ldots t_n)$ rewrites to a term $C[\mathtt{g}(s_1 \ldots s_n)]$, we can analyze the terms rooted at $\mathtt{f}$ and $\mathtt{g}$ and ignore the context for our purposes.

The notion of dependency pair captures this idea by extracting, from every rule, the set of relevant reduction relations.

Definition 1 (Dependency pair) *Let $\mathcal{R}(\mathcal{D}, \mathcal{C}, R)$ be a TRS. If $\mathtt{f}(s_1, \ldots, s_n) \to C[\mathtt{g}(t_1, \ldots, t_m)]$ is a rule from R with $\mathtt{g} \in \mathcal{D}$, then $\mathtt{f}^{\#}(s_1, \ldots, s_n) \to \mathtt{g}^{\#}(t_1, \ldots, t_m)$ is a dependency pair for $\mathcal{R}$.*

The set of all dependency pairs of $\mathcal{R}$ is denoted by $DP_{\mathcal{R}}$.

The trick now is that every infinite derivation contains an infinite number of steps given with a dependency pair. Proving termination of a TRS $\mathcal{R}$ is equivalent to proving that there are no infinite sequences of dependency pairs. We call these sequences *chains*.

Definition 2 (Chain) [Arts and Giesl, 2000; Giesl et al., 2006] *Let $\mathcal{P}, \mathcal{R}$ be two TRS's. A (posibly infinite) sequence of pairs $s_1 \to t_1, s_2 \to t_2, \ldots$ from $\mathcal{P}$ is a $(\mathcal{P}, \mathcal{R})$–chain if there exists a substitution σ with $t_i\sigma \to_{\mathcal{R}} s_{i+1}\sigma$ for all i.*

The gist of the method is in analyzing these chains, which correspond to infinite *minimal* derivations in $\mathcal{R}$, and proving their absence using several techniques in a modular way.

Theorem 1 (Termination Criterion [Arts and Giesl, 2000]**)** *A TRS $\mathcal{R}$ is terminating if and only if no infinite $(DP_{\mathcal{R}}, \mathcal{R})$–chain exists.*

We do not discuss in this section all the available techniques for proving the absence of DP chains. The interested reader can find this information in [Arts and Giesl, 2000; Hirokawa and Middeldorp, 2004; Thiemann, 2007]. Instead, what follows is an intuition of why proving the absence of chains is easier than working with the original TRS directly.

Consider solely, without loss of generality, almost terminating terms[2], i.e., terms whose proper subterms are all terminating, and then mark the root symbol with $^{\#}$ and use $DP_{\mathcal{R}}$ as an extra set of rewriting rules to $\mathcal{R}$. Given such a term $t = \mathtt{f}^{\#}(t_1, , t_n)$, the resulting derivation

$$t \overset{>\varepsilon *}{\to}_{\mathcal{R}} t_0 \overset{\varepsilon}{\to}_{DP_{\mathcal{R}}} t_1 \overset{>\varepsilon *}{\to}_{\mathcal{R}} t_2 \overset{\varepsilon}{\to}_{DP_{\mathcal{R}}} \cdots$$

[2]also called *minimal* non-terminating terms

has been shown to have a direct correspondence to a derivation in $\mathcal{R}$. But the nice thing is that now the rules in $\mathrm{DP}_{\mathcal{R}}$ are used in a topmost only way, as the marked symbols never move below the root. Showing termination of this system $(\mathcal{R} + DP_{\mathcal{R}})$ is equivalent to showing topmost termination of $\mathrm{DP}_{\mathcal{R}}$ relative to $\mathcal{R}$ (informally, relative termination [TeReSe, 2003] of $\mathcal{S}$ w.r.t. $\mathcal{R}$ means that in any infinite $(\mathcal{S}+\mathcal{R})$ derivation one can give only a finite number of steps in $\mathcal{S}$). Both topmost termination and relative termination are easier to prove than proving termination of $\mathcal{S}$ and $\mathcal{R}$ separately. This is one of the advantages of working with the Dependency Pair method: we never have to prove the termination of the original system itself, instead we decompose it into simpler problems.

3 Termination of Narrowing with Dependency Pairs

In this section we introduce the main ideas behind our generalization of the Dependency Pairs method to the narrowing setting. A previous effort in this area is the work of [Nishida, Sakai and Sakabe, 2003], where the DP method is adapted to prove the termination of narrowing in systems that have the so-called *Top Reduced Almost Terminating* (TRAT) property, defined as follows. Given a property P on terms, a term t is said to be a *minimal P term* if t satisfies P but none of the proper subterms of t does. Given a TRS $\mathcal{R}$ and a binary relation $\Rightarrow$ (being $\to_{\mathcal{R}}$ or $\leadsto_{\mathcal{R}}$), an infinite derivation $t \Rightarrow t_1 \Rightarrow t_2 \ldots$ is called *almost terminating* if t is a minimal non–terminating term w.r.t. $\Rightarrow$. An almost terminating derivation $t \Rightarrow t_1 \Rightarrow t_2 \ldots$ is called *top reduced* if it contains a derivation step at the root position. We say that $\Rightarrow$ has the TRAT property if, for every non-terminating term t, there exists a top reduced almost-terminating sequence stemming from one subterm of t.

Let us briefly recall the notion of *context*. A context is a term with several occurrences of a fresh symbol $\square$. In [Nishida, Sakai and Sakabe, 2003] it is proved that every monotone relation has the TRAT property. Since the rewriting relation is monotone (i.e., $t \to_{\mathcal{R}} s$ implies $C[t] \to_{\mathcal{R}} C[s]$), then it has the TRAT property for every TRS $\mathcal{R}$ (cf. [Hirokawa and Middeldorp, 2004, Lemma 1]). In term rewriting this ensures that, in every almost terminating, infinite term rewriting derivation, a rewriting step is given at the root. Unfortunately, the narrowing relation is not monotone: $t \leadsto_{\sigma,\mathcal{R}} s$ does not entail $C[t] \leadsto_{\sigma,\mathcal{R}} C[s]$ but $C[t] \leadsto_{\sigma,\mathcal{R}} (C\sigma)[s]$ instead.

Example 3 [Christian, 1992] *Consider the TRS consisting of the rule* $\mathtt{f}(\mathtt{f}(x)) \to x$*, and the non–linear term* $\mathtt{c}(\mathtt{f}(x), x)$*. Then there does not exist an infinite narrowing derivation for the subterms,* $\mathtt{f}(x)$ *and* x*, whereas* $\mathtt{c}(\mathtt{f}(x), x)$ *is infinitely narrowed without ever performing a narrowing step at the root:*

$$\mathtt{c}(\underline{\mathtt{f}(x)}, x) \leadsto_{\{x \mapsto \mathtt{f}(x')\}} \mathtt{c}(x', \underline{\mathtt{f}(x')}) \leadsto_{\{x' \mapsto \mathtt{f}(x'')\}} \mathtt{c}(\underline{\mathtt{f}(x'')}, x'') \ldots$$

As shown by the above example, in the presence of non linearity the non–monotony of narrowing has undesirable effects for its termination, since *narrexes* can be brought into the context by the substitution computed at the preceding narrowing step, thus causing other terms in the context to grow. This *echoing* effect plays a fatal role in the (non–) termination of narrowing.

There are some classes of TRSs in which narrowing exhibits a monotone or monotone–like behaviour and thus enjoys the TRAT property. [Nishida, Sakai and Sakabe, 2003] considers two such classes: right–linear TRS (w.r.t. linear goals), and constructor systems. We note that these two classes are a particular case of a larger characterization of narrowing termination that we formalized in [Alpuente, Escobar and Iborra, 2008*b*] by the QSRNC (*Quasi stable rigidly normalized condition*), though in [Alpuente, Escobar and Iborra, 2008*b*] we do not make the connection with TRAT explicit.

The inspiration for this work comes from realizing that monotonicity is not really a *necessary* condition for the termination of narrowing, provided the partially computed substitutions do not *echo*, i.e., they do not bring narrexes into the context that might either introduce a term that does not terminate or *echo* again. Let us introduce the idea by means of one example.

Example 4 *Consider the non–linear input call* $\mathtt{c}(\mathtt{f}(x), x)$ *in the non–constructor TRS*

$$\begin{aligned} \mathtt{f}(\mathtt{g}(x)) &\to x \\ \mathtt{g}(x) &\to x \end{aligned}$$

The only possible derivation for this term is finite, whereas the TRS, together with the considered non–linear input term, do not fit in any of the characterizations given for TRAT [Nishida and Miura, 2006; Nishida, Sakai and Sakabe, *2003; Nishida and Vidal, 2008] or any decidable criteria for the termination of narrowing [Alpuente, Escobar and Iborra, 2008*a, b*; Christian, 1992; Hullot, 1980*a*]. Note that the subterm* $\mathsf{g}(x)$ *in the lhs of the first rule is a narrex.*

3.1 The *echoing* problem

Let us start with some lessons learnt from the termination of rewriting that would be good to transfer to the termination of narrowing. In rewriting (and narrowing), if a TRS is not terminating then there must be a minimal non-terminating term. In rewriting such a minimal non-terminating term is rooted by a defined symbol but this is not true for narrowing. As in [Hirokawa and Middeldorp, 2004], let us denote the set of all minimal non-terminating terms w.r.t. rewriting (resp. narrowing) by $\mathcal{T}^{\infty}$ (resp. $\mathcal{T}^{\infty}_{\leadsto}$). The following definition is crucial.

Definition 3 (Echoing terms) *Let $\mathcal{R}$ be a TRS. We define the set of* minimal echoing terms *w.r.t. $\mathcal{R}$, denoted by $\mathcal{T}^{\circlearrowleft}$, as follows: $s \in \mathcal{T}^{\circlearrowleft}$ if*

- *$s \notin \mathcal{T}^{\infty}_{\leadsto}$,*
- *with a fresh binary symbol c and a variable $x \in Var(s)$, $c(s,x) \in \mathcal{T}^{\infty}_{\leadsto}$,*
- *and there is no proper subterm s' of s such that $s' \in \mathcal{T}^{\circlearrowleft}$.*

Now, we provide our key result for the termination of narrowing. We write $s \trianglerighteq t$ to denote that t is a subterm of s, and $s \triangleright t$ if t is a proper subterm of s.

Lemma 1 *Let $\mathcal{R}$ be a TRS. For every term $t \in \mathcal{T}^{\infty}_{\leadsto}$, we have that either*

1. (TOP) *there exists a rewrite rule $l \to r \in \mathcal{R}$, substitutions σ, ρ, a term t', and a non-variable subterm u of r such that $t \overset{>\epsilon}{\leadsto}^{*}_{\rho,\mathcal{R}} t' \overset{\epsilon}{\leadsto}_{\sigma,l\to r} r\sigma \trianglerighteq u$ and $u \in \mathcal{T}^{\infty}_{\leadsto}$;*
2. (HYBRID) *there are terms t', t'', u, substitutions ρ, σ, a position p, and a variable x such that $t \overset{>\epsilon}{\leadsto}^{*}_{\rho,\mathcal{R}} t' \overset{p}{\leadsto}_{\sigma,\mathcal{R}} t''$, $x \in Var(t'|_p)$, $x\sigma \trianglerighteq u$, and $u \in \mathcal{T}^{\infty}_{\leadsto}$;*

3. (ECHOING) *there are terms t', t'', u, substitutions ρ, σ, a position p, and a variable x such that $t \overset{>\epsilon}{\leadsto}^{*}_{\rho,\mathcal{R}} t' \overset{p}{\leadsto}_{\sigma,\mathcal{R}} t''$, $x \in Var(t'|_p)$, $x\sigma \trianglerighteq u$, and $t'|_p, u \in \mathcal{T}^{\circlearrowleft}$.*

Proof. Let D be an infinite narrowing sequence stemming from t. Since all proper subterms of t are terminating for narrowing, D must contain a narrowing step at the root position or there is a narrowing step which computes a substitution that carries a narrex.

- Let us consider the case where there is a narrowing step at the top and cases 2 and 3 do not apply. We consider the first such narrowing step at the root $t \overset{>\epsilon}{\leadsto}^{*}_{\rho,\mathcal{R}} t' \overset{\epsilon}{\leadsto}_{\sigma,l\to r} r\sigma$ where $\sigma = mgu(t', l)$. By assumption, all proper subterms of t' are terminating for narrowing and thus terms brought by the substitution σ are terminating for narrowing. As $r\sigma$ is not terminating for narrowing, it has a subterm $u = r\sigma|_p$ such that u is not terminating for narrowing and, clearly, $p \in \mathcal{P}os(r)$ because all terms brought by σ are terminating for narrowing. Now, by structural induction on r, we easily prove that there is a minimal such u.

- Let us now consider that there is no narrowing step at the top, i.e., case 1 does not apply. We consider two cases:
  - If there is a term t' and a subterm u of t' s.t. $t \overset{>\epsilon}{\leadsto}^{*}_{\rho,\mathcal{R}} t' \trianglerighteq u$ and $u \in \mathcal{T}^{\infty}_{\leadsto}$, then u is introduced by a binding of ρ, since all proper subterms of t are terminating for narrowing.
  - If there is no such u, then the infinite narrowing derivation must be due to a rule $l \to r$ applied infinitely many times, where l, r and all their subterms are terminating for narrowing. This can only happen if there is a proper subterm u of l s.t. $C[u] \leadsto^{*}_{\theta,\mathcal{R}} C'[u']_{p'} \overset{p'}{\leadsto}_{\theta',l\to r} C'\theta'[r\theta']_{p'} \leadsto^{*}_{\theta'',\mathcal{R}} C''[u'']_{p''} \overset{p''}{\leadsto}_{\theta'',l\to r} C''\theta''[r\theta'']_{p''} \cdots$ and $u, u', u'', \ldots \in \mathcal{T}^{\circlearrowleft}$.

□

Informally, the lemma above distinguishes three different kinds of minimal non–terminating terms.

The TOP case is the usual one shared by rewriting and narrowing non–termination; the other two cases are due to non–monotonicity and thus unique to narrowing.

In the pure ECHOING case, the narrowing of an echoing subterm introduces into the context a new echoing subterm that reproduces the process again, as in Example 3.

In the HYBRID echoing case, the reduction of an echoing subterm introduces into the context a minimal non–terminating narrex that spawns an infinite narrowing derivation, as in Example 5 below.

Example 5 *Consider the following TRS:*

$$\mathtt{f}(\mathtt{g}(x)) \to \mathtt{a} \qquad\qquad \mathtt{g}(x) \to \mathtt{g}(x)$$

$\mathtt{g}(x)$ *is in* $c\mathcal{T}^{\infty}_{\leadsto}$, *i.e., it is a minimal non–terminating term for narrowing.* $\mathtt{f}(x) \notin \mathcal{T}^{\infty}_{\leadsto}$, *since only the derivation* $\mathtt{f}(x) \leadsto_{\{x \mapsto \mathtt{g}(x')\}} \mathtt{a}$ *can be proven. However, given a fresh symbol* $\mathtt{c}$, *there is a* HYBRID *infinite narrowing derivation stemming from the term* $\mathtt{c}(\mathtt{f}(x), x) \in \mathcal{T}^{\infty}_{\leadsto}$. *Therefore,* $\mathtt{f}(x) \in \mathcal{T}^{\circlearrowleft}$.

3.2 Narrowing Dependency Pairs

In this section, we develop the notion of narrowing dependency pairs, and provide a sound and complete criterion for the termination of narrowing that is based on analyzing narrowing chains.

The intuition behind our method is as follows. Suppose we split the substitution σ computed by a narrowing step $t \leadsto_{l \to r, \sigma} s$ into two pieces, $\sigma \equiv \sigma_{\upharpoonright l} \uplus \sigma_{\upharpoonright t}$. The $\sigma_{\upharpoonright l}$ part of the substitution has the usual effect of propagating narrexes from the left hand side to the right hand side of the rule. On the other hand, the $\sigma_{\upharpoonright t}$ part is responsible for the echoing of *narrexes* to the context that can fire a new narrowing step. These narrexes come from the subterms of the left hand side of the rule, as in the FullPolicy TRS of Example 3, or from the term being narrowed itself, e.g. when $\mathtt{c}(z, \mathtt{h}(\mathtt{g}(x), z))$ is narrowed to $\mathtt{c}(\mathtt{g}(x), 0)$ by using the rule $\mathtt{h}(y, y) \to 0$ and most general unifier $\{z \mapsto \mathtt{g}(x), y \mapsto \mathtt{g}(x)\}$.

Although the narrexes coming from proper subterms of the original term may cause non–termination, standard (rewriting) termination analyses already cope with them. However, narrexes coming from proper subterms of the lhs of the rules are specific to narrowing, and thus we focus on them in our notion of narrowing dependency pairs.

In order to construct the set of dependency pairs, we not only relate the lhs of each rule with the root–defined subterms occurring in the corresponding rhs, as in standard rewriting DP, but also with its *own* root–defined subterms, i.e., those terms whose root symbol is a defined function. The resulting set of dependency pairs faithfully captures the behaviour of infinite narrowing derivations which incrementally compute an infinite substitution, or more precisely, where the substitution computed by narrowing contains an infinite term.

Notation Let $\mathcal{R}$ be a TRS defined over a signature $\Sigma = \mathcal{D} \uplus \mathcal{C}$. Let $\Sigma^{\#}$ denote the extension of Σ with $\{f^{\#} \mid f \in \mathcal{D}\}$, where $f^{\#}$ is a fresh symbol with the same arity as f. If $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is of the form $\mathtt{f}(s_1, \ldots, s_n)$ with $\mathtt{f}$ a defined symbol, then $\mathrm{t}^{\#}$ denotes the term $\mathtt{f}^{\#}(s_1, \ldots, s_n)$.

The following definition extends the traditional, vanilla DPs with a novel kind of dependency pairs, which we call *ll–dependency pairs.*

Definition 4 (Narrowing Dependency Pair) *Given a TRS $\mathcal{R}$, we have two types of narrowing dependency pairs:*

- *a* lr–dependency pair *(or* standard[1] *DP) of $\mathcal{R}$ is a pair $l^{\#} \to t^{\#}$ where $l \to r \in \mathcal{R}$, $r \trianglerighteq t$, and $root(t) \in \mathcal{D}$.*
- *a* ll–dependency pair *(ll-DP) of $\mathcal{R}$ is a pair $l^{\#} \to u^{\#}$ where $l \to r \in \mathcal{R}$, $l \vartriangleright u$, and $root(u) \in \mathcal{D}$.*

The set of all (narrowing) dependency pairs of $\mathcal{R}$ is denoted by $NDP_{\mathcal{R}}$.

Example 6 *The TRS $\mathtt{f}(\mathtt{f}(x)) \to x$ of Example 3 has no lr–dependency pairs and the single ll–dependency pair $\mathtt{f}^{\#}(\mathtt{f}(x)) \to \mathtt{f}^{\#}(x)$.*

[1]Modern formulations exclude pairs $l^{\#} \to u^{\#}$ when $l \vartriangleright u$. This refinement could be applied to lr-DPs in our definition, but the pair would not be actually discarded, since it is also computed as a ll-DP.

(1)	$\mathtt{filter}^{\#}(\mathtt{pckt}(\mathtt{194.179.1.}x{:}p, dst, \mathtt{new}))$	$\to$	$\mathtt{filter}^{\#}(\mathtt{pckt}(\mathtt{secure}, dst, \mathtt{new}))$
(2)	$\mathtt{filter}^{\#}(\mathtt{pckt}(\mathtt{194.179.1.}x{:}p, dst, \mathtt{new}))$	$\to$	$\mathtt{pckt}^{\#}(\mathtt{secure}, dst, \mathtt{new})$
(3)	$\mathtt{filter}^{\#}(\mathtt{pckt}(\mathtt{158.42.}x.y{:}p, dst, \mathtt{new}))$	$\to$	$\mathtt{filter}^{\#}(\mathtt{pckt}(\mathtt{secure}, dst, \mathtt{new}))$
(4)	$\mathtt{filter}^{\#}(\mathtt{pckt}(\mathtt{158.42.}x.y{:}p, dst, \mathtt{new}))$	$\to$	$\mathtt{pckt}^{\#}(\mathtt{secure}, dst, \mathtt{new})$
(5)	$\mathtt{pckt}^{\#}(\mathtt{10.1.1.1}{:}p, \mathtt{ppp0}, s)$	$\to$	$\mathtt{pckt}^{\#}(\mathtt{123.23.1.1}{:}p, \mathtt{ppp0}, s)$
(6)	$\mathtt{pckt}^{\#}(\mathtt{10.1.1.2}{:}p, \mathtt{ppp0}, s)$	$\to$	$\mathtt{pckt}^{\#}(\mathtt{123.23.1.1}{:}p, \mathtt{ppp0}, s)$
(7)	$\mathtt{filter}^{\#}(\mathtt{pckt}(\mathtt{123.123.1.1}{:}p, dst, \mathtt{new}))$	$\to$	$\mathtt{pckt}^{\#}(\mathtt{123.123.1.1}{:}p, dst, \mathtt{new})$
(8)	$\mathtt{pckt}^{\#}(src, \mathtt{123.123.1.1}{:}p, \mathtt{new})$	$\to$	$\mathtt{pckt}^{\#}(src, \mathtt{10.1.1.1}{:}p, \mathtt{established})$
(9)	$\mathtt{pckt}^{\#}(src, \mathtt{123.123.1.1}{:}p, \mathtt{new})$	$\to$	$\mathtt{pckt}^{\#}(src, \mathtt{10.1.1.2}{:}p, \mathtt{established})$
(10)	$\mathtt{filter}^{\#}(\mathtt{pckt}(src, dst, \mathtt{established}))$	$\to$	$\mathtt{pckt}^{\#}(src, dst, \mathtt{established})$
(11)	$\mathtt{filter}^{\#}(\mathtt{pckt}(\mathtt{eth0}, dst, \mathtt{new}))$	$\to$	$\mathtt{pckt}^{\#}(\mathtt{eth0}, dst, \mathtt{new})$
(12)	$\mathtt{filter}^{\#}(\mathtt{pckt}(\mathtt{194.179.1.}x{:}p, dst, \mathtt{new}))$	$\to$	$\mathtt{pckt}^{\#}(\mathtt{194.179.1.}x{:}p, dst, \mathtt{new})$
(13)	$\mathtt{filter}^{\#}(\mathtt{pckt}(\mathtt{158.42.}x.y{:}p, dst, \mathtt{new}))$	$\to$	$\mathtt{pckt}^{\#}(\mathtt{158.42.}x.y{:}p, dst, \mathtt{new})$
(14)	$\mathtt{filter}^{\#}(\mathtt{pckt}(\mathtt{secure}, dst{:}\mathtt{80}, \mathtt{new}))$	$\to$	$\mathtt{pckt}^{\#}(\mathtt{secure}, dst{:}\mathtt{80}, \mathtt{new})$
(15)	$\mathtt{filter}^{\#}(\mathtt{pckt}(\mathtt{secure}, dst{:}\mathtt{other}, \mathtt{new}))$	$\to$	$\mathtt{pckt}^{\#}(\mathtt{secure}, dst{:}\mathtt{other}, \mathtt{new})$
(16)	$\mathtt{filter}^{\#}(\mathtt{pckt}(\mathtt{ppp0}, dst, \mathtt{new}))$	$\to$	$\mathtt{pckt}^{\#}(\mathtt{ppp0}, dst, \mathtt{new})$
(17)	$\mathtt{pckt}^{\#}(src, \mathtt{123.123.1.1;}p, \mathtt{new})$	$\to$	$\mathtt{natroute}^{\#}(\mathtt{pckt}(src, \mathtt{10.1.1.1}{:}p, \mathtt{established}), \mathtt{pckt}(src, \mathtt{10.1.1.2}{:}p, \mathtt{established}))$

Figure 3.1: Dependency pairs of *FullPolicy*

Example 7 *For the TRS of Example 1 we obtain the narrowing dependency pairs shown in Figure 3.1.*

Recall that our purpose is to prove that there are no infinite narrowing derivations. Since dependency pairs model all function calls in $\mathcal{R}$, this is equivalent to proving that there are no infinite *chains* of narrowing dependency pairs.

For narrowing we consider suitable the following definition of chain. As in [Giesl et al., 2006; Nishida, Sakai and Sakabe, 2003], we assume that different occurrences of dependency pairs are variable disjoint. In the following, $\mathcal{P}$ is usually a set of dependency pairs.

We often omit the ($\mathcal{P}$,$\mathcal{R}$) prefix when referring to narrowing chains when it is clear from the context. The following result establishes the soundness of analyzing narrowing chains.

Lemma 2 *Let $\mathcal{R}$ be a TRS. For every (NDP$_{\mathcal{R}}$,$\mathcal{R}$)–narrowing chain $s_1 \to t_1$, $\ldots, s_n \to t_n$, there exists a narrowing derivation in $\mathcal{R}$ which gives at least one reduction step for every pair in the chain.*

Namely, there are contexts $C_1[\,], \ldots, C_{n+1}[\,]$, positions $p_1, \ldots, p_{n+1}$, terms $u_1, \ldots, u_n$, and substitutions $\tau_1, \ldots, \tau_n, \rho_1, \ldots, \rho_{n-1}$ s.t. $\tau_i = mgu(u_i, s_i)$ for $i \in \{1, \ldots, n\}$, and $C_1[u_1]_{p_1} \overset{p_1}{\leadsto}_{\tau_1,\mathcal{R}} C_2[t_1\tau_1]_{p_2} \overset{>p_2*}{\leadsto}_{\rho_1,\mathcal{R}} C_2\rho_1[u_2]_{p_2} \overset{p_2}{\leadsto}_{\tau_2,\mathcal{R}} C_3[t_2\tau_2]_{p_3}$ $\overset{>p_3*}{\leadsto}_{\rho_2,\mathcal{R}} C_3\rho_2[u_3]_{p_3} \cdots C_n\rho_{n-1}[u_n]_{p_n} \overset{p_n}{\leadsto}_{\tau_n,\mathcal{R}} C_{n+1}[t_n\tau_n]_{p_{n+1}}$.

Proof. By induction on n. The case $n = 0$ is immediate. For $n > 0$, let us consider the suffix of the narrowing sequence that first narrows using the rules from $\mathcal{R}$ and then a dependency pair from the chain, i.e. $t^{\#}_{n-1}\sigma_{n-1} \overset{>\epsilon}{\leadsto}{}^{*}_{\rho_n,\mathcal{R}} u^{\#}_n \overset{\epsilon}{\leadsto}_{\sigma_n, s^{\#}_n \to t^{\#}_n} t^{\#}_n\sigma_n$. By induction hypothesis, we assume there are contexts $C_1[\ ], \ldots, C_n[\ ]$, positions $p_1, \ldots, p_n$, and substitutions $\tau_1, \ldots, \tau_{n-1}$ s.t. $C_1[u_1]_{p_1} \overset{p_1}{\leadsto}_{\tau_1,\mathcal{R}} C_2[t_1\sigma_1]_{p_2} \overset{>p_2}{\leadsto}{}^{*}_{\rho_1,\mathcal{R}} C_2\rho_1[u_2]_{p_2} \overset{p_2}{\leadsto}_{\tau_2,\mathcal{R}} C_3[t_2\sigma_2]_{p_3} \overset{>p_3}{\leadsto}{}^{*}_{\rho_2,\mathcal{R}} C_3\rho_2[u_3]_{p_3} \cdots C_{n-1}\rho_{n-2}[u_{n-1}]_{p_{n-1}} \overset{p_{n-1}}{\leadsto}_{\tau_{n-1},\mathcal{R}} C_n[t_{n-1}\sigma_{n-1}]_{p_n}$. Now, let us consider the two possibilities for $s^{\#}_n \to t^{\#}_n \in NDP_{\mathcal{R}}$.

- If $s^{\#}_n \to t^{\#}_n$ is a vanilla lr-dependency pair, then there is a rule $s_n \to r$ and a position q s.t. $r|_q = t_n$. Therefore, $t^{\#}_{n-1}\sigma_{n-1} \overset{>\epsilon}{\leadsto}{}^{*}_{\rho_n,\mathcal{R}} u^{\#}_n \overset{\epsilon}{\leadsto}_{\sigma_n, s^{\#}_n \to t^{\#}_n} t^{\#}_n\sigma_n$ implies $C_n[t_{n-1}\sigma_{n-1}]_{p_n} \overset{>p_n}{\leadsto}{}^{*}_{\rho_n,\mathcal{R}} C_n\rho_n[u_n]_{p_n} \overset{p_n}{\leadsto}_{\sigma_n, s_n \to r} C_n\rho_n\sigma_n[r\sigma_n]_{p_n}$, where there exists a context C_{n+1} s.t. $C_n\rho_n\sigma_n[r\sigma_n]_{p_n} = C_{n+1}[t_n\sigma_n]_q$ and $p_n \leq q$.

- If $s^{\#}_n \to t^{\#}_n$ is a ll–dependency pair, then there is a rule $s_n \to r$ and a position q s.t. $s_n|_q = t_n$. Therefore, $t^{\#}_{n-1}\sigma_{n-1} \overset{>\epsilon}{\leadsto}{}^{*}_{\rho_n,\mathcal{R}} u^{\#}_n \overset{\epsilon}{\leadsto}_{\sigma_n, s^{\#}_n \to t^{\#}_n} t^{\#}_n\sigma_n$ implies there is a substitution τ_n and a variable $x \in \mathcal{V}ar(u_n) \cap \mathcal{V}ar(C_n\rho_n)$ s.t. $\tau_n(x) = t_n$ and $\tau_n(y) = \sigma_n(y)$ for any other variable y, and $C_n[t_{n-1}\sigma_{n-1}]_{p_n} \overset{>p_n}{\leadsto}{}^{*}_{\rho_n,\mathcal{R}} C_n\rho_n[u_n]_{p_n} \overset{p_n}{\leadsto}_{\tau_n, s_n \to r} C_n\rho_n\tau_n[r\tau_n]_{p_n}$, where there exists a context C_{n+1} and a position $q' \in \mathcal{P}os_x(u_n)$ s.t. $C_n\rho_n\tau_n[r\tau_n]_{p_n} = C_{n+1}[t_n\sigma_n]_{q'}$.

□

Now we are able to show that, whenever there are no infinite narrowing chains, narrowing does terminate.

Theorem 2 (Termination Criterion) *A TRS $\mathcal{R}$ is terminating for narrowing if and only if no infinite ($NDP_{\mathcal{R}}$,$\mathcal{R}$)–narrowing chain exists.*

Proof. The *if* case is straightforward from Lemma 1 and the *only if* case is straightforward from Lemma 2. □

Example 8 *Consider the ll-DP $d \equiv \mathtt{f}^{\#}(f(x)) \to \mathtt{f}^{\#}(x)$ of Example 6. There is a narrowing chain $\mathtt{f}^{\#}(x) \leadsto_{\{x \mapsto \mathtt{f}(x')\},d} \mathtt{f}^{\#}(x') \leadsto_{\{x' \mapsto \mathtt{f}(x'')\},d} \mathtt{f}^{\#}(x'') \cdots$.*

3.3 Automating the method

In order to automate the task of proving the absence of narrowing chains, it would be very convenient to reformulate the problem using only rewriting chains, as it is done in [Nishida, Sakai and Sakabe, 2003; Nishida and Vidal, 2008; Nguyen et al., 2008], since this allows us to reuse existing tools and techniques of the rewriting DP literature. We develop our method inspired by [Nishida and Miura, 2006] but we provide all results without requiring TRAT, which is the main novel contribution of this section. Let us recall the notion of argument filtering.

Definition 5 (Argument Filtering) [Arts and Giesl, 2000] *An argument filtering (AF) for a signature Σ is a mapping π that assigns to every n-ary function symbol $f \in \Sigma$ an argument position $i \in \{1, \ldots, n\}$, or a (possibly empty) list $[i_1, \ldots, i_m]$ of argument positions with $1 \leq i_i < \ldots < i_m \leq n$. The signature Σ_π consists of all function symbols $\mathtt{f}$ s.t. $\pi(\mathtt{f})$ is some list $[i_1, \ldots, i_m]$, where in Σ_π the arity of $\mathtt{f}$ is m. Every AF π induces a mapping from $\mathcal{T}(\Sigma, \mathcal{V})$ to $\mathcal{T}(\Sigma_\pi, \mathcal{V})$:*

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

We extend π to a TRS $\mathcal{R}$ as $\pi(\mathcal{R}) = \{\pi(l) \to \pi(r) \mid l \to r \in \mathcal{R}$ and $\pi(l) \neq \pi(r)\}$. For any argument filtering π and ordering $>$, we define

$$s \geq_\pi t \iff \pi(s) > \pi(t) \textit{ or } \pi(s) \equiv \pi(t)$$

We also define the filtering of a position p w.r.t. a term t as follows. Given a n-ary symbol $f \in \Sigma$ and $i \in \{1, \ldots, n\}$, $\pi(i, f) = j$ if $\pi(f) = [i_1, \ldots, i_j, \ldots, i_k]$, $i_j = i$. Given a term t and a position $p \in \mathcal{P}os(t)$, the filtering of p w.r.t. t is defined as follows:

$$\pi(p, t) = \begin{cases} \epsilon & \textit{if } p = \epsilon \\ \pi(q, t) & \textit{if } p = q.i, i \in \mathbb{N}, \pi(root(t|_q)) = i \\ \pi(q, t).\pi(i, root(t|_q)) & \textit{if } p = q.i, i \in \mathbb{N}, \pi(root(t|_q)) = [i_1, .., i, .., i_k] \end{cases}$$

Example 9 *Consider the TRS of Example 1 and the argument filtering* $\pi(\mathtt{pckt}) = [1,3]$ *and* $\pi(f) = [1,\ldots,ar(f)]$ *for any other* $f \in \Sigma$*. Let us consider the term*

$$t = \mathtt{filter}(\mathtt{pckt}(\mathtt{secure}, dst, \mathtt{new}))$$

its filtered version is

$$\pi(t) = \mathtt{filter}(\mathtt{pckt}(\mathtt{secure}, \mathtt{new}))$$

and the filtering of position 1.3 *is* $\pi(1.3, \pi(t)) = 1.2$ *where* $\pi(1.2, \pi(t))$ *is undefined.*

Definition 6 *Given a TRS* $\mathcal{R}$ *and an AF* π*, we say that* π *is a sound AF for* $\mathcal{R}$ *iff* $\pi(\mathcal{R})$ *is a TRS, i.e., the rhs's of the rules do not contain extra variables not appearing in the corresponding lhs.*

Our main result in this section is Theorem 3 below that relates infinite narrowing $(\mathcal{P}, \mathcal{R})$–chains to infinite rewriting $(\pi(\mathcal{P}), \pi(\mathcal{R}))$–chains. In order to prove this result, we first need two auxiliary lemmata. The first one establishes a correspondence between rewriting derivations in $\mathcal{R}$ and derivations in the filtered TRS $\pi(\mathcal{R})$.

Lemma 3 *Given a TRS* $\mathcal{R}$*, a sound AF* π*, and terms* s *and* t*,* $s \to^*_{\mathcal{R}} t$ *implies* $\pi(s) \to^*_{\pi(\mathcal{R})} \pi(t)$*. Moreover, the derivation in* $\pi(\mathcal{R})$ *uses the same rules in the same order at the corresponding filtered positions (whenever the filtered position exists).*

The next lemma extends the correspondence established in Lemma 3 to narrowing, which can be done only when the original filtered term is ground. The key point is that the correspondence holds *regardless* of the substitution computed by narrowing. It is in fact a (one-way) lifting lemma from narrowing derivations in $\mathcal{R}$ to rewriting sequences in $\pi(\mathcal{R})$.

Lemma 4 *Given a TRS* $\mathcal{R}$ *and a sound AF* π*, let* s *and* t *be terms s.t.* $\pi(s)$ *is ground. Then* $s \rightsquigarrow^*_{\sigma,\mathcal{R}} t$ *implies* $\pi(s) \to^*_{\pi(\mathcal{R})} \pi(t)$*. Moreover, the derivation in* $\pi(\mathcal{R})$ *uses the same rules in the same order at the corresponding filtered positions (whenever the filtered position exists).*

Proof. By the soundness of narrowing, $s \rightsquigarrow^*_{\sigma,\mathcal{R}} t$ implies that $s\sigma \rightarrow^*_{\mathcal{R}} t$, and from Lemma 3 we deduce that $\pi(s\sigma) \rightarrow^*_{\pi(\mathcal{R})} \pi(t)$. Finally $\pi(t)$ must also be ground as π is sound and $\pi(s)$ is ground, hence $\pi(s\sigma) \rightarrow^*_{\pi(\mathcal{R})} \pi(t\sigma) \equiv \pi(s) \rightarrow^*_{\pi(\mathcal{R})} \pi(t)$. □

Let us recall here the standard definition of chain for rewriting.

Definition 7 (Chain) [Arts and Giesl, 2000; Giesl et al., 2006] *Let $\mathcal{P}, \mathcal{R}$ be two TRS's. A (posibly infinite) sequence of pairs $s_1 \rightarrow t_1, s_2 \rightarrow t_2, \ldots$ from $\mathcal{P}$ is a $(\mathcal{P}, \mathcal{R})$–chain if there exists a substitution σ with $t_i\sigma \rightarrow_{\mathcal{R}} s_{i+1}\sigma$ for all i.*

The following result allows us to prove the absence of narrowing chains by analyzing standard rewriting chains. This is very useful because it means that we can reuse all the DP infrastructure available for rewriting. Informally, the idea is that in order to prove termination of narrowing, we must prove termination of rewriting regardless of instantiation. Since narrowing can instantiate the goal, it does not suffice with showing that the lhs is larger than the rhs. One must show that the lhs is larger *than any instance of the rhs.* Thankfully this is only necessary for one pair of the chain, due to the stability under substitutions of the orderings employed.

Theorem 3 *Let $\mathcal{R}$ be a TRS over a signature Σ, $\mathcal{P}$ be a TRS over a signature $\Sigma^{\#}$, and π a sound AF over $\Sigma^{\#}$ s.t. $\pi(t)$ is ground for at least one pair $s \rightarrow t \in \mathcal{P}$ in every ($\mathcal{P}$,$\mathcal{R}$)–narrowing chain. If there exists no infinite $(\pi(\mathcal{P}), \pi(\mathcal{R}))$–chain, then there exists no infinite ($\mathcal{P}$,$\mathcal{R}$)–narrowing chain.*

Proof. By contradiction. Suppose there is such an infinite chain, and therefore an infinite narrowing derivation exists of the form

$$t_1 \overset{\varepsilon}{\rightsquigarrow}_{\sigma_1,\mathcal{P}} u_1 \rightsquigarrow^*_{\sigma_2,\mathcal{R}} t_2 \overset{\varepsilon}{\rightsquigarrow}_{\sigma_2,\mathcal{P}} u_2 \rightsquigarrow^*_{\sigma_3,\mathcal{R}} \cdots$$

As some $\pi(u_i)$ is ground by assumption, by Lemma 4 there is a rewriting derivation in $\pi(\mathcal{R})$ of the form

$$\pi(u_i) \rightarrow^*_{\pi(\mathcal{R})} \pi(t_1) \xrightarrow{\varepsilon}_{\pi(\mathcal{P})} \pi(u_{i+1}) \xrightarrow{\varepsilon}_{\pi(\mathcal{P})} \cdots$$

which by assumption must be finite. Note that every step given with a rule from $\mathcal{P}$ is given in the filtered derivation too, as those steps are given at the root position. By Lemma 4, finiteness of the filtered derivation implies that the narrowing derivation cannot have infinite $\mathcal{P}$ steps, and we reach a contradiction which proves the theorem. □

The following straightforward consequence of Theorems 2 and 3 characterizes termination of narrowing as a rewriting problem.

Corollary 1 *Let $\mathcal{R}$ be a TRS over a signature Σ, and π a sound AF over $\Sigma^{\#}$ s.t. $\pi(t)$ is ground for at least one pair $s \to t \in NDP_{\mathcal{R}}$ in every $(NDP_{\mathcal{R}}, \mathcal{R})$–narrowing chain.*

If there exists no infinite $(\pi(NDP_{\mathcal{R}}), \pi(\mathcal{R}))$–chain, then narrowing terminates in $\mathcal{R}$.

3.3.1 Extending the DP framework to narrowing

We switch now our attention to the DP *framework* of [Thiemann, 2007]. In this framework a DP *problem* is a tuple $(\mathcal{P}, \mathcal{R})$ of two TRSs, $\mathcal{R}$ and $\mathcal{P}$, where initially $\mathcal{P} = DP(\mathcal{R})$. If there is no associated infinite chain, we say that the problem is *finite*. Termination methods are then formulated as *DP processors* that take a DP problem and return a new set of DP problems. A DP processor is *sound* if the input problem is finite whenever all the output problems are.

The remaining of this section shows how to recast the problem of termination of narrowing in the DP *framework*. In the usual style [Nishida and Vidal, 2008; Nguyen et al., 2008], we show here how to adapt a few of the most important DP processors, and then give one that transforms a narrowing DP problem into a rewriting one, which allows us to use any of the existing DP processors for termination of rewriting. We speak of *narrowing DP problems* to distinguish them from the standard ones. A narrowing DP problem has the same components as an ordinary one, i.e., it is a tuple of two TRSs $\mathcal{P}$ and $\mathcal{R}$ where initially $\mathcal{P} = NDP_{\mathcal{R}}$.

3.3.2 Dependency Graph processor

The following definition adapts the standard notion of dependency graph to our setting by considering narrowing dependency pairs instead of vanilla DPs.

Definition 8 (Dependency Graph) *Given a (narrowing) DP problem $\langle \mathcal{P}, \mathcal{R} \rangle$ its (resp. narrowing) dependency graph is the directed graph where the nodes are the elements of $\mathcal{P}$, and there is an edge from $s \to t \in \mathcal{P}$ to $u \to v \in \mathcal{P}$ if $s \to t$, $u \to v$ is a (resp. narrowing) chain from $\mathcal{P}$.*

The theorem below establishes that the narrowing dependency graph of a narrowing DP problem is equal to the dependency graph of the rewriting DP problem defined by the same TRS and DP set.

Theorem 4 *Given a narrowing DP problem $\langle \mathcal{P}, \mathcal{R} \rangle$, its narrowing dependency graph is the same as the dependency graph of the rewriting DP problem defined by $\langle \mathcal{P}, \mathcal{R} \rangle$.*

Proof. Let NG be the narrowing dependency graph for $\langle \mathcal{P}, \mathcal{R} \rangle$, and G be the dependency graph for the rewriting DP problem. Both graphs contain the elements of $\mathcal{P}$ as nodes.

($\Rightarrow$) We show that every edge in NG is also in G. From the definition of narrowing chain, there is an edge from $l^\# \to r^\#$ to $s^\# \to t^\#$ if there exist terms u_1, u_2 and substitutions σ_1, σ_2 s.t. $u_1 \overset{\epsilon}{\rightsquigarrow}_{\sigma_1, l^\# \to r^\#} r^\# \sigma_1 \overset{\geq\epsilon}{\rightsquigarrow}^*_{\rho_1, \mathcal{R}} u_2 \overset{\epsilon}{\rightsquigarrow}_{\sigma_2, s^\# \to t^\#} t^\# \sigma_2$.

The definition of rewriting chain requires that there exists a substitution σ s.t. $r^\# \sigma \to^*_{\mathcal{R}} s^\# \sigma$. By Hullot's lifting lemma such a σ exists and is equal to or less general than $\sigma_1 \rho_1 \sigma_2$.

($\Leftarrow$) We show that for every edge in G is in NG too. There is an edge in G from $l^\# \to r^\#$ to $s^\# \to t^\#$ if there exists a substitution σ s.t. $\sigma r^\# \to^*_{\mathcal{R}} \sigma s^\#$. By the soundness of narrowing $r^\# \sigma \rightsquigarrow^*_{id, \mathcal{R}} s^\# \sigma$. If we take $u_1 \equiv l^\# \sigma$, then the following narrowing chain exists: $u_1 \overset{\varepsilon}{\rightsquigarrow}_{\sigma, l^\# \to r^\#} r^\# \sigma \rightsquigarrow^*_{id, \mathcal{R}} s^\# \sigma \overset{\varepsilon}{\rightsquigarrow}^*_{\sigma, s^\# \to t^\#} t^\# \sigma$. $\square$

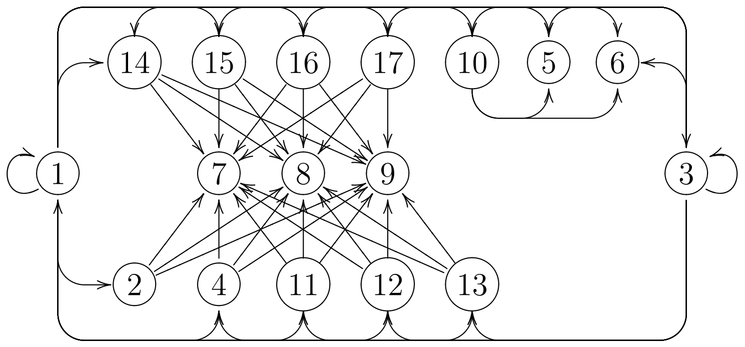

Figure 3.2: Estimated dependency graph of *FullPolicy*

It is well known that computing the exact dependency graph is undecidable and thus several approximations [Giesl et al., 2006] are used to compute an *estimated* dependency graph which includes the exact graph. The following approximation is commonly used.

Definition 9 (Estimated Dependency Graph) [Giesl, Thiemann and Schneider-Kamp, 2005] *Let $\langle \mathcal{P}, \mathcal{R} \rangle$ be a DP problem. Let $\mathsf{CAP}_{\mathcal{R}}(t)$ be the result of replacing*[2] *all the proper subterms of t with a defined root symbol by a fresh variable, and $\mathsf{REN}(t)$ the linearization of t (replacing all ocurrences of a non linear variable with independent fresh variables). The nodes of the estimated dependency graph (EDG) are the pairs of $\mathcal{P}$ and there is an edge from $s^{\#} \to t^{\#}$ to $u^{\#} \to v^{\#}$ iff $\mathsf{REN}(\mathsf{CAP}_{\mathcal{R}}(t))$ and u are unifiable.*

Example 10 *For the problem of Example 1 and the set of DPs obtained in Example 7, the EDG is shown in Figure 3.2.*

For finite TRSs, infinite chains show up as cycles in the dependency graph[3]. We can analyze separately every chain, that is, every cycle in the dependency graph. This is accomplished by the Dependency Graph processor.

Theorem 5 (Dependency Graph Processor) [Giesl, Thiemann and Schneider-Kamp, 2005] *For a DP problem $\langle \mathcal{P}, \mathcal{R} \rangle$, let Proc be the processor that returns*

[2]This function was first defined for approximating loops in dependency graphs in [Alpuente, Falaschi and Vidal, 1994], where it is called $\overset{\circ}{t}$.

[3]The converse does not hold, not every cycle corresponds to an infinite chain.

$$\begin{tikzcd} 1 \arrow[loop left] \arrow[r, bend left] & 3 \arrow[l, bend left] \arrow[loop right] \end{tikzcd}$$

Figure 3.3: Filtered *FullPolicy*

problems $\{\langle\mathcal{P}_1, \mathcal{R}\rangle, \ldots, \langle\mathcal{P}_n, \mathcal{R}\rangle\}$*, where* $\mathcal{P}_1, \ldots, \mathcal{P}_n$ *are the sets of nodes of every cycle in the estimated dependency graph. Proc is sound.*

Proof.

Let EG be the EDG, G the exact one, and NG the narrowing one. We prove that every chain generates a cycle, by showing that every pair of DPs in a chain corresponds to an edge in a cycle in EG. By definition this holds for G, and NG. Hence one must show that $G \subseteq EG$, which has already been done in e.g. [Giesl, Thiemann and Schneider-Kamp, 2005], and by Theorem 4 we have that $NG \equiv G$, and the claim follows. □

Example 11 *In the graph obtained in the EDG of Example 10, the only cycle consists of (1) and (3). Thus the dependency graph processor deletes all the other dependency pairs, and returns the problems* $\{$ *(*$\{$*(1),(3)*$\}$*,* $\mathcal{R}$*), (*$\{$*(1)*$\}$*,* $\mathcal{R}$*), (*$\{$*(3)*$\}$*,* $\mathcal{R}$*)*$\}$ *corresponding to the graph shown in Figure 3.3.*

3.3.3 Reduction Pair processor

The next processor we adapt is the standard reduction pair processor. Let us introduce the standard notion of reduction pair.

Definition 10 (Reduction Pair) *A reduction pair* $(\succeq, \succ)$ *consists of a quasi-rewrite ordering* $\succeq$ *and an ordering* $\succ$ *with the following properties: (i)* $\succ$ *is closed under substitutions and well founded, and (ii)* $(\succ \circ \succeq) \subseteq \succ$*.*

For a narrowing DP problem $\langle\mathcal{P}, \mathcal{R}\rangle$, this processor tries to find a reduction pair $(\succeq, \succ)$ and a suitable filtering π s.t. all the filtered $\mathcal{R}$-rules are weakly decreasing w.r.t. $\succeq$, and all filtered $\mathcal{P}$ pairs are weakly or strictly decreasing. For any TRS $\mathcal{P}$ and relation $\succ$, let $\mathcal{P}_\succ = \{s \to t \mid s \succ t\}$.

$$\begin{tikzcd}
 & 2 \arrow[dr] & \\
1 \arrow[ur] \arrow[loop left] & & 3 \arrow[ll] \arrow[loop right]
\end{tikzcd}$$

Figure 3.4: Dependency Graph

Theorem 6 (Reduction Pair processor) *Let $(\mathcal{P},\mathcal{R})$ be a narrowing DP problem s.t. $\mathcal{P}$ is a cycle*[4]*, $(\succeq, \succ)$ be a reduction pair, and π be an argument filtering s.t. $\pi(t)$ is ground for at least one pair $s \to t \in \pi(\mathcal{P})$. The following processor is sound. $Proc_\pi(\mathcal{P},\mathcal{R})$ returns*

- $\{(\mathcal{P} \setminus \mathcal{P}_{\succ_\pi}, \mathcal{R})\}$ *if* $\mathcal{P}_{\succ_\pi} \cup \mathcal{P}_{\succeq_\pi} = \mathcal{P}$, $\mathcal{P}_{\succ_\pi}$ *is not empty, and* $\mathcal{R}_{\succeq_\pi} = \mathcal{R}$*;*
- $\{(\mathcal{P},\mathcal{R})\}$ *otherwise.*

Proof. By [Arts and Giesl, 2000, Theorem 11] the constraints guarantee that there is no infinite rewriting chain from $\mathcal{P}$ in $\mathcal{R}$, which implies no infinite rewriting chain from $\pi(\mathcal{P})$ in $\pi(\mathcal{R})$. Now, by Theorem 3, there is no infinite narrowing chain from $\mathcal{P}$ in $\mathcal{R}$. □

Note that it is not enough to consider all the pairs in a strongly connected component (SCC) at once, as it is commonly done in rewriting, and that we consider cycles instead. The reason is that the condition of Theorem 3, groundness of one DP rhs per chain (cycle), would not be ensured when working with SCCs instead, as the following example shows.

Example 12 *Consider a TRS $\mathcal{R}$ with the Dependency Graph of Figure 3.4. Our dependency graph processor decomposes this problem into three subproblems corresponding to the cycles $\{1\}$, $\{3\}$ and $\{1,2,3\}$. A SCC approach would consider only the last one. Suppose we did indeed use SCCs. The reduction pair processor defined above can synthetize a filtering π_2 s.t. the rhs of (2) is ground and an ordering s.t. (3) can be oriented strictly; upon doing so it will remove (3) of the DP problem, thus leaving only $\{1,2\}$. This eliminates two*

[4]Note that this requirement is easily fulfilled by running the dependency graph processor first.

cycles at once, {3} and {1,2,3}. But this is unsound: we cannot eliminate the cycle in {3} as we have not yet provided an argument filtering π_3 s.t. the rhs of (3) is ground and there is a suitable ordering.

3.3.4 Argument Filtering processor

We claim that it is straightforward to adapt most of the standard DP processors in order to deal with the grounding AF requirement, and due to lack of space we will present only one more processor, which can be used to transform a narrowing DP problem into an ordinary one. Afterwards, any existing DP processor for rewriting becomes applicable.

Theorem 7 (Argument Filtering Processor) *Let $(\mathcal{P}, \mathcal{R})$ be a narrowing DP problem s.t. $\mathcal{P}$ is a cycle, and π be an argument filtering s.t. $\pi(t)$ is ground for at least one pair $s \to t \in \pi(\mathcal{P})$. Then, $Proc_\pi(\mathcal{P}, \mathcal{R}) = \{(\mathcal{P}_\pi, \pi(\mathcal{R}))\}$, where $\mathcal{P}_\pi$ is defined as $\mathcal{P}_\pi = \{\pi(l) \to \pi(r) \mid l \to r \in \mathcal{P}, l \not\rhd r\}$. $Proc_\pi$ is a sound narrowing DP processor.*

Proof. Given the narrowing DP problem $P \equiv (\mathcal{P}, \mathcal{R})$, let $Proc_\pi(P) = P'$. If P' is finite then there are no infinite rewriting $(\mathcal{P}_\pi, \pi(\mathcal{R}))$–chains, which means that there are no infinite $(\pi(\mathcal{P}), \pi(\mathcal{R}))$–chains, as the set of discarded pairs $\pi(\mathcal{P}) \setminus \mathcal{P}_\pi$ cannot produce infinite rewriting chains [Dershowitz, 2003]. By Theorem 3, this implies that there are no infinite $(\mathcal{P},\mathcal{R})$–narrowing chains. $\square$

Finally, we include the subterm refinement in the AF processor as it can be the case that the rhs of a DP becomes a subterm of the lhs after the filtering.

Example 13 *The set of narrowing DP problems resulting of Example 11 can be solved by using the AF processor to transform them into rewriting problems.*

- *$(\{(1)\}, \mathcal{R})$ For this problem soundness requires that $\pi(\texttt{pckt}) = [1, 3]$. Using the identity for all other symbols, we get the following (rewriting) DP problem that is finite, as one can easily check with a modern termination tool implementing the DP method such as Aprove [Giesl et al.,*

2004], or Mu-Term [Alarcón et al., 2007]:
$(\{\texttt{filter}^{\#}(\texttt{pckt}(\texttt{194.179.1}.x\!:\!p, \texttt{new})) \to \texttt{filter}^{\#}(\texttt{pckt}(\texttt{secure}, \texttt{new})\}, \mathcal{R})$

- *($\{(3)\}$, $\mathcal{R}$) In this case, we proceed in a similar way, and the same AF π allows us to transform the current subproblem into a finite (rewriting) DP problem.*

- *($\{(1),(3)\}$, $\mathcal{R}$) Finally, by using the same AF π, we get a finite DP problem.*

This finally proves that the FullPolicy TRS is terminating for narrowing.

4 Implementation

We have implemented a tool named *Narradar* for the automatic proving of the termination of narrowing, available publicly at

`http://safe-tools.dsic.upv.es/narradar`

The tool offers a web interface accepting TRSs in the standard TPDB format from the termination problem database[1], and implements the approach illustrated in this thesis to prove the termination of narrowing, relying on the rewriting termination tool AproVe [Giesl et al., 2004] to solve the derived rewriting problems. If Narradar succeeds a short report of the proof is presented. Otherwise it fails with a "don't know" response. In both cases a diagrammatic log of the attempted proof is presented.

Narradar itself takes roughly 1000 lines of Haskell, while another 1000 lines are invested in a more general library that implements the key elements of Term Rewriting systems, such as terms, substitutions, rules, matching and unification. In the following, we describe the parts of this library essential to Narradar, as well as Narradar itself.

4.1 The TRS library

We describe now an encoding in Haskell of the fundamental notions of term rewriting such as terms, variables, substitutions, matching and unification. This library serves as a basis upon which more complex tools such as Narradar itself can be built.

[1]http://www.lri.fr/ marche/tpdb/format.html

4.1.1 Terms and Rules

Terms are usually viewed as labelled trees built from a signature and a set of variables. Usually branches contain function symbols with an arity of one or more, whereas leafs can be either variables or function symbols of arity zero. But this is a rather naive view of terms. We can be interested in branches or leafs of other kinds, such as for instance a hole, if we are modelling a context, or bottom, if we are modelling pointed terms. Therefore in the TRS library the Haskell datatype for terms is open in the sense of OO inheritance, i.e., further constructors can be added at any point. We use the encoding for open data types described by [Swierstra, 2008], where a term is the recursive closure of a coproduct of functors.

```
newtype Term f = In {out::f (Term f)}
```

As Swierstra puts it, you can think of the type argument f as the *term signature*, the list of the constructors allowed in our terms. Less intuitively, $Term$ is just a fixed-point type-level operator which takes as parameter a functor and ties the knot over it. All this is better understood with an example.

Two suitable functors are `Var`, for variables, and `T`, for function symbols:

```
data Var a = Var {idV:: Maybe String, unique::Int}
             deriving (Eq, Ord)
instance Functor Var where fmap f (Var u i) = Var u i

data T id a = T {idT::id, subTerms::[a]} deriving (Eq, Ord)
instance Functor (T id) where fmap f (T id tt) = T id (map f tt)
```

The Var functor simply ignores its type argument, whereas in the T functor – which is additionally parameterized with the type of the symbols – the argument is used to type the subterms.

We also define the type of Holes, suitable later for building contexts:

```
data Hole a = Hole Int
```

Now that we have three functors, we can combine them by means of a coproduct of functors, which is defined as follows

```
data (f :+: g) a = Inl (f a) | Inr (g a)
```

The type `(f :+: g)` is equivalent to the usual `Either` datatype in Haskell, it just works with functors instead of values. A coproduct of two functors is a functor itself too.

```
instance (Functor f, Functor g) => Functor (f :+: g) where
  fmap f (Inl e1)  = Inl (fmap f e1)
  fmap f (Inr e2)  = Inr (fmap f e2)
```

We can now construct terms as follows.

```
-- t = f(0,x)
t :: Term (T String :+: Var)
t = In (Inl (T "f" [In(Inr(Var (Just "x") 1))]))
```

Of course it is not very convenient to manipulate terms in this way. Swierstra introduces a notion of subtyping between functors that leads to a generic injection function.

```
class (Functor sub, Functor sup) => sub :<: sup where
  inj :: sub a -> sup a
  prj :: sup a -> Maybe (sub a)
```

`inj` is our generic injection function and now we can define smart constructors for our functors as follows:

```
inject :: g :<: f => g(Expr f) -> Expr f
inject    = In . inj

var u i   = inject(Var u i)
term i tt = inject(T i tt )
hole      = inject . Hole
```

GHC infers the following types for our constructors

```
var  :: (Var  :<: f) => Int -> Maybe (String) -> Term f
term :: (T id :<: f) => id -> [Term f] -> Term f
hole :: (Hole :<: f) => Int -> Term f
```

And we can define rules as follows.

```
type Rule f = RuleG (Term f)
data RuleG a = !a :-> !a deriving (Eq, Show)
instance Functor RuleG where fmap f (l:->r) = f l :-> f r
--Lexicographic ordering
instance (Eq (RuleG a),Ord a) => Ord (RuleG a) where
  compare (l1 :-> r1) (l2 :-> r2) = case compare l1 l2 of
                                      EQ -> compare r1 r2
                                      x  -> x
lhs,rhs :: forall t. RuleG t -> t
lhs (l :-> _) = l
rhs (_ :-> r) = r

infix 1 :->
```

As an example let us define addition of natural numbers using the standard Peano encoding.

```
s x = term "s" [x]
z   = term "0" []
x   = var (Just "x") 0
y   = var (Just "y") 1
x +: y = term "+" [x,y]

peano = [ z +: x :-> x, s x +: y :-> s (x +: y)]

-- Two example terms, with their term signature fixed
sx = s x :: Term (Var :+: T String)
sz = s z :: Term (Var :+: T String)
```

But we still haven't shown the instances that give meaning to the (:<:) operator. Swierstra presents a simple encoding with only three instances designed to avoid issues with the GHC typechecker. We have found this encoding lacking in practice and more expressive encodings are definable. However there is no single best option, e.g. it is not possible to show transitivity to the GHC typechecker, so currently one must interact with the typechecker to find an encoding which 'just works' for the problem at hand. We happily gloss over this issue here.

The main way to define operations of terms is by means of a polymorphic fold over their structure.

```
foldTerm :: (f a -> a) -> Term f -> a
foldTerm f (In t) = f (fmap (foldTerm f) t)
```

We must provide `foldTerm` with a polymorphic *algebra* or interpretation. Polymorphic in the sense that, since the data constructors are not known in advance, as usually when defining a fold over a closed datatype, we need to resort to an open polymorphic algebra encoded by type class encoded.

Let us see a simple example, a function for pretty printing a term. We define a `Ppr` type class and suitable instances. The implementation of the instances makes use of the standard Haskell pretty printing library [Hughes, 1995].

```
class Functor f => Ppr f where pprF :: f Doc -> Doc
instance Ppr Var where pprF (Var Nothing u)   = char 'u' <> int u
                       pprF (Var (Just id) u) = text id
instance Show id => Ppr (T id) where
  pprF (T n []) = text (show n)
  pprF (T n [x,y])
      | not (any isAlpha $ show n) = x <+> text (show n) <+> y
  pprF (T n tt) = text (show n) <>
                  parens (cat$ punctuate comma tt)
```

We need to throw in an instance for (:+:) in order to make coproducts printable too:

```
instance (Ppr a, Ppr b) => Ppr (a:+:b) where
    pprF (Inr x) = pprF x
    pprF (Inl y) = pprF y
```

Finally, the ppr function will print any term, as long as there is a `Ppr` instance for all the constructors in it.

```
ppr :: Ppr f => Term f -> Doc
ppr = foldTerm pprF

instance Ppr f => Show (Term f) where show = render . ppr
```

A nice property of this way of defining folds is that our traversals stay open. So if we add a new term type, as we will do later for contexts, it suffices to make the new term type an instance of our polymorphic algebra and the traversal will work for it too.

Let us see another example, a function to calculate the size of a term (when seen as a tree).

```
class (Functor f, Foldable f) => Size f where
    sizeF :: f Int -> Int
instance (Functor f, Foldable f) => Size f where
    sizeF f = 1 + sum f

sizeTerm = foldTerm sizeF
```

The `Foldable` class for applicative functors [Mcbride and Paterson, 2008] is very convenient in this case, since we can define the interpretation of `sizeF` once and for all for any applicative functor[2]. Since all our contructors for terms are applicative by definition, this is a great thing. But first we need to make clear that coproducts are instances of `Foldable`, as well as the constructors in our term signature:

[2]Note that GHC forces us to ask for undecidable instances for the generic Size instance, since the constraint is no smaller than the head of the instance.

```
instance (Foldable f, Foldable g) => Foldable (f :+: g) where
    foldMap f (Inl x) = foldMap f x
    foldMap f (Inr x) = foldMap f x

instance Foldable Var where foldMap f _ = mempty
instance Foldable (T id) where foldMap f (T _ tt) = foldMap f tt
```

There are more nifty things we can do using `Foldable`. For instance, we can retrieve all the subterms of a term with a `Foldable` signature.

```
subterms, properSubterms :: (Functor f, Foldable f) =>
                            Term f -> [Term f]
subterms (In t) = concat (subterms <$> toList t)
properSubterms = tail . subterms
```

`toList` is a standard operation on `Foldable`s which retrieves the list of all the elements inside.

```
%This is not really code
toList :: Foldable f => f a -> [a]
```

We can use `subterms` to define many useful operations on terms.

```
vars :: (Var :<: s, Foldable s, Functor s) =>
        Term s -> [Var (Term s)]
vars t = nub [ v | u <- subterms t, Just v@Var{} <- [open u]]

collect :: (Foldable f, Functor f) =>
           (Term f -> Bool) -> Term f -> [Term f]
collect pred t = [ u | u <- subterms t, pred u]

isLinear :: (Var :<: s, Foldable s, Functor s) => Term s -> Bool
isLinear t = length(nub vars_t) == length vars_t
    where vars_t = vars t
```

In order to pattern match on `Term` values we used an `open` combinator, which fails if the term has not the right type. It is similar in concept to a `cast` operation for dynamic typing.

```
open :: (g :<: f) => Term f -> Maybe (g (Term f))
open (In t) = prj t
```

Here is an operation to extract the symbol at the root position of a term (if any) which relies on `open`.

```
rootSymbol :: (T id :<: f) => Term f -> Maybe id
rootSymbol t | Just (T root _) <- open t = Just root
             | otherwise = Nothing
```

In other cases, it is more convenient to use `prj` directly.

```
collectIds :: (T id :<: f) => Term f -> [id]
collectIds = foldTerm f where
    f t | Just (T id ids) <- prj t = id : concat ids
        | otherwise = []

foldTermM :: (Monad m, Traversable f) =>
             (f a -> m a) -> Term f -> m a
foldTermM f (In t) = f =<< mapM (foldTermM f) t
```

`foldTermM` is a generalization for monadic algebras, but only available if the constructor signature is `Traversable` by applicative functors.

`foldTerm` and `foldTerM` provide a bottom-up traversal on a `Term` value. We can also define top-down traversals, but then the algebra provided must operate on `Terms` exclusively. Top-down traversals are implemented by `foldTermTop`.

```
foldTermTop :: Functor f =>
               (f (Term f) -> f(Term f)) -> Term f -> Term f
foldTermTop f (In t)= In (foldTermTop f `fmap` f t)
```

We will see an example of a top-down traversal in next section when we define a function to annotate subterms with their position.

Working with positions

Positions are specified as lists of Ints, where the empty list corresponds to the root position. The expression (t ! p) denotes the subterm of t at position p.

```
type Position = [Int]

(!) :: Foldable f => Term f -> Position -> Term f
In t ! (i:ii) = (toList t !! i) ! ii
t    ! []     = t
```

We can annotate all the components of a term with their position as follows. First we declare the WithNote functor, which allows to attach a note to a base functor f.

```
newtype WithNote note f a = Note (note, f a) deriving (Show)
instance Functor f  => Functor (WithNote note f)  where
    fmap f (Note (p, fx))   = Note (p, fmap f fx)

note :: Term (WithNote note f) -> note
note (In (Note (note,_))) = note

dropNote :: Functor f => Term (WithNote note f) -> Term f
dropNote = foldTerm f where f (Note (note,t)) = In t
```

Annotating with positions is a top-down fold over a term previously extended to contain notes with a bottom-up fold. Remember that top-down folds cannot really modify the structure of a term, this needs to be done by a bottom-up fold. But it is awkward to work with the positions in a bottom-up way, so that's why we split the work in two folds.

The AnnotateWithPos class defines the algebra for extending a term's signature to carry notes of positions. The notes are initialized in the appropriate default way for every constructor in the signature.

```
class (t :<: f) => AnnotateWithPos t f where
```

```
  annotateWithPosF :: t (Term (WithNote Position f)) ->
                          Term (WithNote Position f)
instance (T id :<: f) => AnnotateWithPos (T id) f where
  annotateWithPosF (T n tt) =
     In$ Note ([], (inj$ T n [In (Note (i:p, t))
                             | (i, In(Note (p,t))) <- zip [0..] tt]))
instance (t  :<: f) => AnnotateWithPos t f where
    annotateWithPosF t = In $ Note ([], inj t)
instance ((a :+: b) :<: f, AnnotateWithPos a f, AnnotateWithPos b f)
    => AnnotateWithPos (a :+: b) f where
  annotateWithPosF (Inr x) = annotateWithPosF x
  annotateWithPosF (Inl y) = annotateWithPosF y
```

To complete the process, the top-down fold `mergePosF` propagates the positions from the top using `appendPos`.

```
annotateWithPos :: AnnotateWithPos f f =>
                   Term f -> Term (WithNote Position f)
annotateWithPos = foldTermTop mergePosF . foldTerm annotateWithPosF
  where
   mergePosF (Note (p,t)) = Note (p, fmap (appendPos p) t)
   appendPos p (In (Note (p', t'))) = In (Note (p++p', t'))
```

Before leaving, we will throw in some additional instances that will be useful when manipulating annotated terms.

```
instance Traversable f => Traversable (WithNote note f) where
 traverse f (Note (p, fx)) = (Note . (,) p) <$> traverse f fx
instance Foldable f => Foldable (WithNote note f) where
  foldMap f (Note (_p,fx)) = foldMap f fx
instance (Functor f, Eq (Term f)) => Eq (Term (WithNote note f))
  where t1 == t2 = dropNote t1 == dropNote t2
instance (Functor f, Ord (Term f)) => Ord (Term (WithNote note f))
  where t1 `compare` t2 = compare (dropNote t1) (dropNote t2)
```

```
instance (Show note, Ppr t) => Ppr (WithNote note t) where
  pprF (Note (p,t)) = pprF t <> char '_' <> text (show p)
instance IsVar f => IsVar (WithNote note f) where
  isVarF (Note (_,t)) = isVarF t
  uniqueIdF (Note (_,t)) = uniqueIdF t
```

Binary operations on Open Datatypes

So far we have seen how to consume terms individually. The definition of binary functions over terms is based on zipping. The `zipTermF` type class below allows us to zip together the subterms of terms which have a matching shape. If the shape does not match, then zipTerm fails in the corresponding monad.

```
class Functor f => ZipTerm f where
  zipTermF  :: Monad m => (a -> b -> m c)  -> f a -> f b -> m (f c)
  zipTermF_ :: Monad m => (a -> b -> m ()) -> f a -> f b -> m ()
  zipTermF_ f t u = zipTermF f t u >> return ()

zipTermM :: (ZipTerm f, Monad m) =>
          (Term f -> Term f -> m c) -> Term f -> Term f -> m (f c)
zipTermM f (In t) (In u) = zipTermF f t u
```

We need an instance to distribute zipping over the coproduct of functors, and further instances for each of our term constructor types.

```
instance (ZipTerm a, ZipTerm b) => ZipTerm (a :+: b) where
    zipTermF  f (Inl x) (Inl y) = Inl `liftM` zipTermF f x y
    zipTermF  f (Inr x) (Inr y) = Inr `liftM` zipTermF f x y
    zipTermF  f _       _       = fail "zipTermF"
    zipTermF_ f (Inl x) (Inl y) = zipTermF_ f x y
    zipTermF_ f (Inr x) (Inr y) = zipTermF_ f x y
    zipTermF_ f _       _       = fail "zipTermF"
instance Eq id => ZipTerm (T id) where
    zipTermF  f (T s1 tt1) (T s2 tt2) = do
```

```
      unless(s1 == s2 && length tt1 == length tt2) $ fail "zipTermF"
      T s1 `liftM` zipWithM f tt1 tt2
    zipTermF_ f (T s1 tt1) (T s2 tt2) = do
      unless(s1 == s2 && length tt1 == length tt2) $ fail "zipTermF"
      zipWithM_ f tt1 tt2
instance ZipTerm Var where
    zipTermF f (Var u1 i) (Var u2 _) | u1 == u2 = return (Var u1 i)
                                     | otherwise = fail "zipTermF"
```

Note that this definition requires that the terms not only have the same shape, but also are in the same coproduct. If we want to zip terms of different coproducts we must first `reinject` the smaller coproduct into the larger one.

```
reinject :: (f :<: g) => Term f -> Term g
reinject = foldTerm inject
```

From `zipTermM` we can reconstruct the standard rendition of `zip`.

```
zipTerm :: ZipTerm f => Term f -> Term f -> Maybe (f(Term f, Term f))
zipTerm = zipTermM (*) where a * b = return (a,b)
```

In the following sections there are several examples on how to use `zipTerm`, as our definitions of matching and unification rely heavily on it.

4.1.2 Substitutions

A substitution is a mapping from variables to terms. This simple statement is a little bit more tricky when working with open data types, since the user of the library might wish to add new constructors for variables, which we cannot anticipate yet. To stay flexible, we define the fold `IsVar`.

```
class Functor f => IsVar f where
    isVarF    :: f x -> Bool
    uniqueIdF :: f x -> Maybe Int

instance IsVar Var where isVarF _ = True
```

```
                         uniqueIdF(Var _ u) = Just u
instance (IsVar a, IsVar b) => IsVar (a:+:b) where
    isVarF (Inr x) = isVarF x
    isVarF (Inl y) = isVarF y
    uniqueIdF (Inr x) = uniqueIdF x
    uniqueIdF (Inl x) = uniqueIdF x

instance Functor otherwise => IsVar otherwise where
    isVarF _ = False; uniqueIdF _ = Nothing

isVar :: IsVar f => Term f -> Bool
isVar = foldTerm isVarF

uniqueId :: IsVar f => Term f -> Maybe Int
uniqueId = foldTerm uniqueIdF
```

For instance, this is how we use this class to extract all the variables of a term.

```
vars' :: (IsVar s, Ord (Term s), Foldable s, Functor s) =>
         Term s -> [Term s]
vars' = nub . collect isVar

isGround :: (IsVar f, Ord(Term f), Foldable f, Functor f) =>
            Term f -> Bool
isGround = null . vars'
```

Our substitutions are isomorphic to an association list.

```
type Subst f = SubstG (Term f)
newtype SubstG a = Subst {fromSubst:: Map.Map Key a}
    deriving (Eq, Functor, Show)
```

As representation of the domain, we are going to use not the uniques, but the variable contructors themselves. This is to stay friendly when showing substitutions to the user. However, we will employ a fast Ord instance based on the underlying unique to keep things reasonably efficient.

```
data Key where KeyTerm :: (Ppr f, IsVar f) => Term f -> Key
instance Eq Key  where t1 == t2      = keyUnique t1 == keyUnique t2
instance Ord Key where
    compare k1 k2 = keyUnique k1 ‘compare‘ keyUnique k2
instance Show Key where showsPrec _ (KeyTerm t) = (show(ppr t) ++)

keyUnique (KeyTerm t) = (‘fromMaybe‘ uniqueId t)
       (error "used a non variable in the domain of a substitution")
```

The creator of `Key` values is responsible for ensuring that only terms which are really variables are used in the domain of a substitution. In order to build a substitution, there are several options. One can handle the keys as values of type `Var`, or as values of type `Term`. Both seem reasonable enough, since while the former is more correct, the latter is more convenient. Therefore we provide a polymorphic function `mkSubst` that accepts both alternatives.

```
class MkSubst a f | a -> f where mkSubst :: a -> Subst f
instance IsVar fs =>  MkSubst [(Var h, Term fs)] fs where
  mkSubst vv_tt = mkSubst [ (In (Var u i), t)
                           | (Var u i,t) <- vv_tt]
instance (Ppr k, IsVar k, IsVar fs) =>
            MkSubst [(Term k, Term fs)] fs where
  mkSubst = mkSubst . map (first KeyTerm)
          . filter (isJust.uniqueId.fst)
instance IsVar f => MkSubst [(Key, Term f)] f where
  mkSubst = mkSubst . Map.fromList
instance IsVar f => MkSubst (Map Key (Term f)) f where
  mkSubst = normalize . Subst

normalize (Subst map) =
    Subst $ Map.filterWithKey
```

```
                (\k t -> Just(keyUnique k) /= uniqueId t) map
```

We want `Subst` to be an ADT, therefore the `Key` data type must not be exported, and neither the `Subst` constructor. The application of a substitution is defined as follows[3].

```
applySubst :: (IsVar f, f :<: fs) =>
              Subst fs -> Term f -> Term fs
applySubst s t = foldTerm (applySubstF s) t

applySubstF :: (IsVar f, f :<: fs) =>
               Subst fs -> f (Term fs) -> Term fs
applySubstF s t
  | isNothing uid = inject t
  | Just i <- uid = fromMaybe (inject t) $ lookupKey s i
  where uid = uniqueIdF t

lookupSubst :: IsVar g => Subst f -> Term g -> Maybe (Term f)
lookupSubst s t | Just i <- uniqueId t = lookupKey s i
                | otherwise = Nothing

lookupKey :: Subst f -> Int -> Maybe (Term f)
lookupKey (Subst s) i =
        Map.lookup (KeyTerm (inV (Var Nothing i))) s

inV :: Var t -> Term Var
inV (Var n i) = In (Var n i)
```

Substitutions are monoids with the empty substitution as the identity element and left biased union as the associative operator.

```
instance IsVar f => Monoid (Subst f) where
```

[3]in order to keep this text simple, we omit the code to rename free variables in the codomain of the substitution

```
  mempty  = Subst mempty
  mappend (Subst map1) s2@(Subst map2) =
    mkSubst (map2 ‘mappend‘ map1’)
      where Subst map1’ = mkSubst (applySubst s2 <$> map1)
```

The last piece we need is a function to extend a substitution with a new binding. We can build this easily on top of the Monoid instance we defined, by creating a singleton substitution and appending the remaining.

```
insertSubst :: (Ppr k, IsVar k, IsVar fs) =>
               Term k -> Term fs -> Subst fs -> Subst fs
insertSubst v t sigma = mkSubst [(v,t)] ‘mappend‘ sigma
```

4.1.3 Matching

We are now in a position to implement matching already. We say that a term t matches a term l if there is a substitution σ such that $t = l\sigma$. We define this in Haskell by means of a Match class.

A first naive attempt

```
class (Functor f1, Functor f2) => Match f1 f2 where
      matchF :: (f1:<:g, f2:<:g, Match g g, IsVar g) =>
                f1(Term g) -> f2(Term g) -> Maybe (Subst g)


class Match f f => Matchable f; instance Match f f => Matchable f
match :: (IsVar f, Matchable f) =>
         Term f -> Term f -> Maybe (Subst f)
match (In t) (In u) = matchF t u
```

This says that we can match two terms of different shapes as long as they are built with the same coproduct. We need to use a multi parameter type class for the first time so far. There are two constraints added to the matchF type signature; we need them on the term signature because match is recursive in the instances we will define now. Let’s start with the instances for the coproduct. We need to decompose coproducts on both sides.

```
instance (Match c a, Match d a) => Match (c :+: d) a where
    matchF (Inl x) y = matchF x y
    matchF (Inr x) y = matchF x y
instance (Match a c, Match a d) => Match a (c :+: d) where
    matchF x (Inl y) = matchF x y
    matchF x (Inr y) = matchF x y
```

We need a third instance to disambiguate in the case we encounter a co-product on both sides.

```
instance (Match a c, Match a d, Match b c, Match b d) =>
         Match (a :+: b) (c :+: d) where
    matchF (Inl x) (Inl y) = matchF x y
    matchF (Inr x) (Inr y) = matchF x y
    matchF (Inl x) (Inr y) = matchF x y
    matchF (Inr x) (Inl y) = matchF x y
```

That was all the boilerplate needed, and now we can define the meaningful instances. In particular, a variable matches anything, any other constructor matches only terms with the same shape, as long as the subterms match.

```
instance (Foldable f, ZipTerm f) => Match f f where
    matchF t u = fold <$> zipTermF match t u

instance Functor f => Match Var f where
    matchF v u = Just $ mkSubst [(v, inject u)]
instance Match Var Var where
    matchF v u = Just $ mkSubst [(v, inject u)]
```

But since the `Var` instance makes use of `inject`, it cannot be typed unless we show the compiler that the functor bound to `f` is really a member of the term signature. That is, we need to add a constraint `f :<: g` to the `Var` instance, where `g` is the term signature, but `g` is not in scope here. We therefore need to add the constraint to the `matchF` type signature in the declaration of the `Match` class. However, this does not work as expected, because then we need to show that this constraint is fulfilled in recursive calls to `matchF`, and here we run into problems in the coproduct instances. The GHC typechecker cannot deduce `c :<: g` from the knowledge that `c:+:d :<: g`. And that is for good reason: the type checker is no theorem prover.

Splitting the work A solution to this problem is to add the term signature as a parameter of the `Match` type class, so that it is in scope later when we need it. This works but it introduces noise and is less than ideal. We want to hide this to the user as much as we can. To accomplish that, we introduce two new classes. As a boon this also allows us to control the order in which the decomposition of the functor products is performed. We will decompose first the lhs in the two params class `MatchL`. At this point if we have a var in the lhs we can already match. Next we decompose the rhs in the three params class `MatchR`. Finally, the user will see the two params class `Match` we defined before.

Note that for terms with only function symbols and variables, the second decomposition step is unnecessary. It might be useful however in the event that the user of the library introduces some other constructor in the signature with a particular matching behaviour.

```
class (f:<:g) => MatchL f g where
    matchL :: f(Term g) -> g(Term g) -> Maybe (Subst g)
instance ((c:+:d) :<: a, MatchL c a, MatchL d a) =>
  MatchL (c:+:d) a where
    matchL (Inl x) y = matchL x y
    matchL (Inr x) y = matchL x y
instance (Var:<:g, IsVar  g) => MatchL Var g where
    matchL v@Var{} t = Just $ mkSubst [(v, In t)]
instance MatchR f g g => MatchL f   g where matchL = matchR

class (f1:<:g, f2:<:g) => MatchR f1 f2 g where
    matchR :: f1 (Term g) -> f2 (Term g) -> Maybe (Subst g)
instance ((c:+:d):<:g, MatchR a c g, MatchR a d g) =>
```

```
  MatchR a (c :+: d) g where
    matchR x (Inl y) = matchR x y
    matchR x (Inr y) = matchR x y
instance (f:<:gs, g:<:gs, IsVar gs, Match gs gs, Match f g) =>
  MatchR f g gs where matchR = matchF
```

4.1.4 Unification

We follow the same scheme for the implementation of unification. Below we show how to adapt the standard Martelli&Montanari algorithm to open datatypes. In this case there is not much point in establishing an ordering in the product decomposition, since we can't just stop on the presence of a variable in the lhs; we need to fully decompose the rhs too. But even so, splitting the task in two type classes makes it easier because there are less ambiguities to deal with than when one tries to define all the decomposition instances in the same type class.

```
class    UnifyL f f => Unifyable f
instance UnifyL f f => Unifyable f

class (f :<: g) => UnifyL f g where
    unifyL :: (MonadPlus m, MonadEnv g m) =>
              f (Term g) -> g (Term g) -> m ()
instance (UnifyL a c, UnifyL b c, (a:+:b):<:c) =>
  UnifyL (a :+: b) c where
    unifyL (Inl x) y = unifyL x y
    unifyL (Inr x) y = unifyL x y
instance UnifyR f g g => UnifyL f g where unifyL = unifyR

class (f1 :<: g, f2 :<: g) => UnifyR f1 f2 g where
    unifyR :: (MonadPlus m, MonadEnv g m) =>
              f1 (Term g) -> f2 (Term g) -> m ()
instance (UnifyR c a g, UnifyR c b g, (a:+:b):<:g) =>
  UnifyR c (a :+: b) g where
```

```
    unifyR x (Inl y) = unifyR x y
    unifyR x (Inr y) = unifyR x y
instance (f1:<:g, f2:<:g, Unifyable g, Ppr g, Unify f1 f2) =>
  UnifyR f1 f2 g where unifyR = unifyF
```

That was all the boilerplate code for the decomposition, now we can define the `Unify` class, where the real work is done.

```
class Unify f1 f2 where
  unifyF :: (f1 :<: g, f2:<:g, Unifyable g, Ppr g,
              MonadPlus m, MonadEnv g m) =>
            f1(Term g) -> f2(Term g) -> m ()
instance Unify Var t where unifyF v t = varBind (inV v) (inject t)
instance Unify t Var where unifyF t v = varBind (inV v) (inject t)
instance Unify Var Var where
    unifyF v@(Var n i) w@(Var _ j)
        | i == j    = return ()
        | otherwise = do
              mb_t <- readVar (inV v)
              case mb_t of
                Nothing -> varBind (inV v) (inject w)
                Just t  -> unify' t (inject w)
instance (Foldable f, ZipTerm f) => Unify f f where
    unifyF t u = zipTermF_ unify' t u


unify' :: (MonadPlus m, MonadEnv f m, Unifyable f) =>
          Term f -> Term f -> m ()
unify' (In t) (In u) = unifyL t u
```

Unification makes use of an environment monad which provides operations for binding a variable, reading the contents of a variable, applying the environment to an entire term, and finally retrieving the environment. The concrete implementation of this monad, which we do not show here, can be made on top of a State monad, or also on top of a monad with references such as the IO monad.

```
class (Functor m, Monad m, IsVar f) => MonadEnv f m | m -> f where
    varBind :: (IsVar g, Ppr g) => Term g -> Term f -> m ()
    readVar :: IsVar g => Term g -> m (Maybe (Term f))
    apply   :: (IsVar g, g:<:f) => Term g -> m (Term f)
    getEnv  :: m (Subst f)

instance (IsVar f, Functor m, MonadState (Subst f) m) => MonadEnv f m
  where
    varBind = (modify.) . insertSubst
    apply t = get >>= \sigma -> return (applySubst sigma t)
    getEnv  = get
    readVar = gets . flip lookupSubst

runEnv m = execStateT m mempty
```

And finally this is the `unify` function for the end user.

```
unify :: (MonadPlus m, Unifyable f, IsVar f) =>
         Term f -> Term f -> m (Subst f)
unify t u = runEnv (unify' t u)
```

4.1.5 TRSs

It is useful to introduce the type of TRSs, which package a set of rules together with a description of the signature used by those rules. For that we introduce the `Signature` datatype, carrying the `Set`s of constructors and defined symbol, as well as the arity function. All this information is extracted from a set of `Rule`s in the natural way.

```
data Signature id = Sig { constructorSymbols :: Set id
                        , definedSymbols     :: Set id
                        , arity              :: Map id Int}
     deriving (Show, Eq)

getSignatureFromRules :: (T id :<: f, Ord id, Foldable f) =>
```

```
                          (id -> id) -> [Rule f] -> Signature id
getSignatureFromRules mkLabel rules =
      Sig{arity= Map.fromList
                  [(mkLabel f,length tt)
                       | l :-> r <- rules, t <- [l,r]
                       , Just (T f tt) <- map open (subterms t)]
         , definedSymbols     = Set.fromList dd
         , constructorSymbols =
             Set.fromList $ map mkLabel $
             [root | l :-> r <- rules
                   , t <- subterms r ++ properSubterms l
                   , Just root <- [rootSymbol t]] \\ dd
         }
    where dd = nub [ root | l :-> _ <- rules
                          , let Just root = rootSymbol l]
```

Signatures are `Monoids` in the obvious way.

```
instance Ord id => Monoid (Signature id) where
    mempty  = Sig mempty mempty mempty
    mappend (Sig c1 s1 a1) (Sig c2 s2 a2) =
      Sig (mappend c1 c2) (mappend s1 s2) (mappend a1 a2)
```

Additionally we define `TRSC`, an alias for a set of constraints that a constructor signature must fulfill in order to be used in a TRS. Namely, it must support `Vars`, it must support function symbols which must also be in `Ord`, it must be matchable and unifiable, and finally it must also be `Traversable`, which in turn implies `Foldable`. From this point, every time we inspect a set of rules inside a TRS, GHC will be able to deduce that all these constraints hold. This is very handy to alleviate the number of constraints we must include later in types of functions that manipulate TRSs.

```
class    (T id :<: f, Ord id, Var :<: f, IsVar f, ZipTerm f
         , Ord(Term f), Traversable f, Unifyable f, Matchable f
         , AnnotateWithPos f f) => TRSC id f
instance (T id :<: f, Ord id, Var :<: f, IsVar f, ZipTerm f
         , Ord(Term f), Traversable f, Unifyable f, Matchable f
         , AnnotateWithPos f f) => TRSC id f
data TRS id f where
    TRS :: TRSC id f => [Rule f] -> Signature id -> TRS id f

tRS            :: (TRSC id f) => [Rule f] -> TRS id f
rules          :: TRS id f -> [Rule f]
sig            :: TRS id f -> Signature id

tRS rules       = TRS rules (getSignatureFromRules id rules)
rules (TRS r _) = r
sig   (TRS _ s) = s
```

TRSs are monoidal too.

```
instance TRSC id f => Monoid (TRS id f) where
   mempty = TRS mempty mempty
   mappend (TRS r1 _) (TRS r2 _) =
     TRS rr (getSignature rr) where rr = r1 `mappend` r2
```

For convenience, we are going to define an overloaded `getSignature` version which works on anything that has an associated Signature, including sets of rules and TRSs.

```
class SignatureC a id | a -> id where
    getSignature :: a -> Signature id
instance (T id :<: f, Ord id, Foldable f) =>
         SignatureC [Rule f] id
    where getSignature = getSignatureFromRules id
instance SignatureC (TRS id f) id where getSignature = sig
```

Finally, we introduce the familiar `isConstructor` and `isDefined` predicates, as well as the `getArity` function.

```
isDefined, isConstructor :: (T id :<: f, Ord id) =>
                             TRS id f -> Term f -> Bool
isConstructor trs t = (‘Set.member‘ constructorSymbols (sig trs))
                        ‘all‘ collectIds t
isDefined           = (not.) . isConstructor

getArity :: (Show id, Ord id) => Signature id -> id -> Int
getArity Sig{arity} f = (‘fromMaybe‘ Map.lookup f arity)
    (error("getArity: symbol " ++ show f ++ " not in signature"))
```

4.2 Modelling Dependency Pairs

Until now we have described the relevant parts of the TRS library. We have seen that the TRS library provides terms with open signatures. Let us begin by fixing the constructors signature of the terms used in Narradar.

```
type Basic   = T String :+: Var
type BasicId = T Identifier :+: Var
```

We are going to need the ability to mark function symbols as dependency pairs, and for that reason we make use of terms with identifiers of type `Identifier` instead of `String`. Such terms are denoted by the signature `BasicId`.

```
data Identifier = IdFunction String | IdDP String
    deriving (Eq, Ord)
instance Show Identifier where
    show (IdFunction f) = f; show (IdDP n) = n ++ "#"

markDPSymbol (IdFunction f) = IdDP f
markDPSymbol f = f
unmarkDPSymbol (IdDP n) = IdFunction n
unmarkDPSymbol n = n

markDP, unmarkDP :: (T Identifier :<: f) => Term f -> Term f
```

```
markDP t
  | Just (T (n::Identifier) tt) <- open t
    = term (markDPSymbol n) tt
  | otherwise = t
unmarkDP t
  | Just (T (n::Identifier) tt) <- open t
    = term (unmarkDPSymbol n) tt
  | otherwise = t

unmarkDPRule, markDPRule :: (T Identifier :<: f) =>
                            Rule f -> Rule f
markDPRule   = fmap   markDP
unmarkDPRule = fmap unmarkDP
```

We project TRSs with `String` function symbols as follows.

```
mkTRS :: [Rule Basic] -> TRS Identifier BasicId
mkTRS rr = TRS rules' (getSignatureFromRules id rules') where
    rules' = fmap2 (foldTerm mkTIdF) rr :: [Rule BasicId]

fmap2 :: (Functor f, Functor g) => (a -> b) -> f(g a) -> f(g b)
fmap2 = fmap . fmap

class  (Functor f, Functor g) => MkTId f g where
    mkTIdF :: f (Term g) -> Term g
instance (T Identifier :<: g) => MkTId (T String) g where
    mkTIdF (T f tt) = term (IdFunction f) tt
instance (MkTId f1 g, MkTId f2 g) => MkTId (f1 :+: f2) g where
    mkTIdF (Inl x) = mkTIdF x; mkTIdF (Inr x) = mkTIdF x
instance (a :<: g) => MkTId a g where
    mkTIdF t = inject(fmap reinject t)
```

We define DPs and DP problems as follows.

```
type DP f = Rule f
type Problem f = Problem_ (TRS Identifier f)
data Problem_ a = Problem  ProblemType a a deriving (Eq,Show)
data ProblemType = Rewriting | Narrowing deriving (Eq, Show)
```

So a `Problem` carries two TRSs, one for the rules and one for the DPs. And we have two types of problems, for rewriting and for narrowing. The signature of a problem is the sum of the signatures of the rules and the DPs.

```
instance SignatureC (Problem f) Identifier where
  getSignature (Problem _ trs@TRS{} dps@TRS{}) =
                                       sig trs ‘mappend‘ sig dps
```

We define functions `getPairs` and `getLPairs` to extract the right(standard) and left(narrowing) dependency pairs.

```
getPairs :: TRS Identifier f -> [DP f]
getPairs trs@TRS{} =
    [ markDP l :-> markDP rp | l :-> r <- rules trs,
                               rp <- collect (isDefined trs) r]

getLPairs :: TRS Identifier f -> [DP f]
getLPairs trs@TRS{} = [ markDP l :-> markDP lp
                            | l :-> _ <- rules trs
                            , lp <- properSubterms l
                            , isDefined trs lp]

getNPairs :: TRS Identifier f -> [DP f]
getNPairs trs = getPairs trs ++ getLPairs trs
```

4.2.1 Proof searching

The solution of a DP problem is reached after a proof search using the available problem processors. This proof search is recorded via the functor `ProofT`.

```
data ProofF f s k =
    And     {procInfo::ProcInfo, problem::Problem f, subProblems::[k]}
  | Or      {procInfo::ProcInfo, problem::Problem f, subProblems::[k]}
  | Success {procInfo::ProcInfo, problem::Problem f, res::s}
  | Fail    {procInfo::ProcInfo, problem::Problem f, res::s}
  | DontKnow{procInfo::ProcInfo, problem::Problem f}
  | MPlus k k
  | MZero
    deriving (Show)
```

A proof tree with terms of constructor signature `f` contains conjunctive and disjunctive branches with subproblems of type `k`, and success, failure, and don't know leaves which may contain a description of type `s`. The `MPlus` branches and `MZero` leaves are used later to build a `MonadPlus` instance (MonadPlus models a simplified version of computational search). The datatype `ProcInfo` identifies the available problem processors.

```
data ProcInfo = AFProc (Maybe AF)
              | DependencyGraph
              | Polynomial
              | External ExternalProc
              | NarrowingP
              deriving (Eq, Show)
data ExternalProc = MuTerm | Aprove | Other String
      deriving (Eq, Show)
```

We define a predicate `isSuccess` to find out whether a proof has been completed or not.

```
isSuccessF :: ProofF f s Bool -> Bool
isSuccessF Fail{}         = False
isSuccessF Success{}      = True
isSuccessF DontKnow{}     = False
isSuccessF (And _ _ ll)   = and ll
isSuccessF (Or  _ _ ll)   = or ll
isSuccessF MZero          = False
isSuccessF (MPlus p1 p2) =  p1 || p2
```

`ProofF` is a functor on the branches of the search tree. But this time we will automatically derive its instances (as well as the ones for `Problem`, which is a functor too). Even more, we require it to be traversable by applicative functors, and we will derive the `Traversable` instances too. [4]

As it is a functor, we can automatically extract a *free monad* out of `ProofF` [Swierstra, 2008]. We will build our solvers on top of this monad. Let us introduce first the standard definition of a free monad.

```
-- This is the standard encoding of Free Monads, see
--  http://comonad.com/reader/2008/monads-for-free
data Free f a = Impure (f (Free f a)) | Pure a
instance Functor f => Monad (Free f) where
    return          = Pure
    Pure a    >>= f = f a
    Impure fa >>= f = Impure (fmap (>>= f) fa)
```

All we need to declare a free monad `Proof` out of our functor `ProofF` is a simple type synonym declaration!

```
type Proof f s a      = Free (ProofF f s) a

type ProblemProof s f = Proof f s (Problem f)
```

We are going to work with monadic actions on DP `Problems`. In free monad speak, a `Pure` value is an actual value, wrapped inside the monad, whereas an `Impure` value encodes some kind of effect. The constructors of the Functor used to instantiate the free monad determine what kind of impure actions are available. Later, some interpretation function translates these hints of impure effects into actual effects. In our case, there is no translation involved: what

[4] The instances are derived via Template Haskell thanks to the package Data.Derive. The code to invoke the derivations is included at the end of this chapter.

we are interested in is the very ProofF object constructed during the search of the solution. What the free monad construction does in our case is to extend our ProofF trees with a new type of leaves, Pure leaves. Binding in the Proof monad corresponds to mapping a function to *only the* Pure *leaves* and then joining the tree returned to the rest of the tree[5]. In general, the type of values in Pure leaves can be anything, so we can for instance construct a ProofF tree with Strings in the Pure leaves. At the moment however (this may change later when we want to *display* our proof trees), we are interested only in Proof trees with Problem*s* at the Pure leaves. ProblemProof is a type synonym to stress that fact.

Before continuing, let us introduce smart constructors to lift the ProofF contructors to the Proof monad.

```
success = ((Impure.).) . Success
failP   = ((Impure.).) . Fail
andP    = ((Impure.).) . And
orP     = ((Impure.).) . Or
choiceP = (Impure.)    . MPlus
dontKnow= (Impure.)    . DontKnow
```

We also need a monad transformer version of Proof, since some of our solvers must invoke an external tool via the IO monad.

```
--   (built upon Luke Palmer control-monad-free hackage package)
newtype FreeT f m a =
    FreeT {unFreeT :: m (Either a (f (FreeT f m a)))}

editEither l r = either (Left . l) (Right . r)
conj f = FreeT . f . unFreeT

instance (Functor f, Functor m) => Functor (FreeT f m) where
    fmap f = conj $ fmap (editEither f ((fmap.fmap) f))
```

[5]in other words, recall that $x \gg= f = join(fmap\ f\ x)$

```
instance (Functor f, Monad m) => Monad (FreeT f m) where
    return  = FreeT . return . Left
    m >>= f = FreeT $ unFreeT m >>= \r ->
        case r of
             Left x   -> unFreeT $ f x
             Right xc -> return . Right $ fmap (>>= f) xc
instance (Functor f) => MonadTrans (FreeT f) where
    lift = FreeT . liftM Left

type ProofT      f s m a = FreeT (ProofF f s) m a
type ProblemProofT s m f = ProofT f s m (Problem f)
runProofT x = unwrap x
```

We define some operations on free monads, as well as a generic projection between free monad transformers and regular free monads.

```
foldFree :: Functor f => (a -> b) -> (f b -> b) -> Free f a -> b
foldFree pure _    (Pure   x) = pure x
foldFree pure imp  (Impure x) = imp (fmap (foldFree pure imp) x)

foldFreeT :: (Traversable f, Monad m) =>
             (a -> m b) -> (f b -> m b) -> FreeT f m a -> m b
foldFreeT p i m = do
   r <- unFreeT m
   case r of
     Left   x -> p x
     Right fx -> join (liftM i (mapM (foldFreeT p i) fx))

unwrap :: (Traversable f, Monad m) => FreeT f m a -> m(Free f a)
unwrap = foldFreeT (return . Pure) (return . Impure)

wrap :: (Functor f, Monad m) => Free f a -> FreeT f m a
wrap  = FreeT . foldFree (return . Left)
                         (return . Right . fmap FreeT)
```

By means of `foldFree` and `foldFreeT` we can lift `ProofF` folds to the associated free monad. Note that `foldFreeT` requires that the functor `f` is traversable, as it needs to sequence a monadic action through the structure of a `f` value. This is why we required traversability for `ProofF`.

As an example, this is how we lift the `isSuccessF` predicate to the `Proof` and `ProofT` monads.

```
isSuccess :: Proof f s a -> Bool
isSuccess  = foldFree  (const False) isSuccessF

isSuccessT :: Monad m => ProofT f s m a -> m Bool
isSuccessT = foldFreeT (const $ return False) (return.isSuccessF)
```

Finally, the `Proof` and `ProofT` free monads (transformer) are also in `MonadPlus`, since proof finding involves search.

```
%Not valid GHC code
instance MonadPlus (Proof f s) where
    mzero         = Impure MZero
    p1 ‘mplus‘ p2 = if isSuccess p1 then p1 else choiceP p1 p2

instance Monad m => MonadPlus (ProofT f s m) where
    mzero         = FreeT $ return $ Right MZero
    p1 ‘mplus‘ p2 = FreeT $ do
                        s1  <- runProofT p1
                        if isSuccess s1 then unFreeT(wrap s1)
                           else do s2  <- runProofT p2
                                   unFreeT (wrap(choiceP s1 s2))
```

In the next section, after defining a few processors, we will show how to use `ProofF` free monad to build a minilanguage of proof tactics.

4.2.2 The Dependency Graph processor

This processor involves building a graph out of the Dependency Pairs and computing all the cycles it contains. For representing the graphs we use Martin

Erwig's graph library, [Erwig, 2001], using an adapted version of Liu and Wang algorithm to compute the cycles in a graph.

```
--  "A new way to enumerate cycles in graph" - Hongbo Liu, Jiaxin Wang
cycles :: Graph gr => gr a b -> [[Node]]
cycles gr = (nub  . map (sort . nub . map fst))
            (concatMap liuwang [[(n,n)] | n <- nodes gr]) where
 liuwang path = [ path ++ [closure]
                  | let closure = (tpath, hpath)
                  , closure `elem` edges gr] ++
                      concatMap liuwang
                       [ path++[(tpath,n)]
                        | n <- suc gr tpath
                        , n /= hpath
                        , (tpath,n) `notElem` path]
     where tpath = (snd.last) path
           hpath = (fst.head) path
```

The approximations `ren` and `cap` are straightforward to define with the machinery introduced so far.

```
ren :: (Var :<: f, Traversable f) => Term f -> Term f
ren t = runSupply (foldTermM f t) where
    f t | Just Var{} <- prj t = var Nothing <$> next
        | otherwise           = return (inject t)


cap :: TRS Identifier f -> Term f -> Term f
cap trs@TRS{} t | Just (T (s::Identifier) tt) <- open t
                = term s [if isDefined trs t'
                            then var Nothing i else t'
                                | (i,t') <- [0..] `zip` tt]
                | otherwise = t
```

We employ the well known `Supply` monad defined on top of `State` to obtain an infinite supply of fresh names in `cap`.

```
class MonadSupply i m | m -> i where next :: m i

newtype Supply i a = Supply {runSupply_ :: State [i] a}
  deriving (Functor, Monad, MonadSupply i)

instance MonadSupply e (State [e]) where
  next = do
      elems <- get
      put (tail elems)
      return (head elems)

runSupply :: (Num i, Bounded i, Enum i) => Supply i a -> a
runSupply m = evalState (runSupply_ m) [0..]
```

With these tools, constructing the dependency graph processor is easy. It is provided by the monadic `cycleProcessor` function below. First one computes the estimated dependency graph (EDG) making use of the `ren` and `cap` functions, and then `cycles` computes the cycles in the EDG, which are then used to build a number of subproblems under a conjunction branch.

```
cycleProcessor :: Problem f -> ProblemProof String f
cycleProcessor problem@(Problem typ trs@TRS{} dps)
  | null cc= success DependencyGraph problem
             ("We need to prove termination for all the cycles."
           ++ "There are no cycles, so the system is terminating")
  | otherwise =
      andP DependencyGraph problem
         [ return $ Problem typ trs (tRS$ select (rules dps) ciclo)
           | ciclo <- cc]
    where cc = cycles $ getEDG trs (rules dps)

getEDG :: TRS Identifier f -> [DP f] -> G.Gr () ()
getEDG trs@TRS{} dps = G.mkUGraph [0.. length dps - 1]
                          [ (i,j) | (i,_:->t) <- zip [0..] dps
```

```
                                   , (j,u:->_) <- zip [0..] dps
                                   , inChain t u]
    where inChain t u = isJust (unify u (ren $ cap trs $ t))

select :: (Ord t, Enum t, Num t) => [a] -> [t] -> [a]
select xx ii = go 0 xx (sort ii) where
          go _ [] _ = []
          go _ _ [] = []
          go n (x:xx) (i:ii) | n == i = x : go (succ n) xx ii
                             | otherwise = go (succ n) xx (i:ii)
```

`select` is a function to retrieve consecutive indexes from a list. It must satisfy the following property.

```
propSelect xx ii = map (xx!!) ii' == select xx ii'
  where types = (xx::[Int], ii::[Int])
        ii'   = filter (< length xx) (map abs ii)
```

We just defined the dependency graph processor as a monadic action in the `ProofF` free monad. In the incoming sections we will see how to combine it with other proof processors in order to build a proof tactic.

4.2.3 The Argument Filtering processor

Before defining the AF processor we need to introduce the `AF` datatype for argument filterings, as well as some algebra to manipulate them.

```
newtype AF = AF {fromAF:: Map Identifier (Set Int)}
    deriving (Eq, Ord)


singletonAF :: Identifier -> [Int] -> AF
cut         :: Identifier -> [Int] -> AF -> AF
cutAll      :: [(Identifier, [Int])] -> AF -> AF
lookupAF    :: Monad m => Identifier -> AF -> m [Int]
```

```
fromListAF :: [(Identifier,[Int])] -> AF
toListAF   :: AF -> [(Identifier,[Int])]

singletonAF id ii  = AF (Map.singleton id (Set.fromList ii))
cut id ii (AF m) = AF $ Map.insertWith (flip Set.difference) id
                         (Set.fromList ii) m
cutAll xx af     = foldr (uncurry cut) af xx
lookupAF id (AF m) = maybe (fail "not found")
                           (return.Set.toList)
                           (Map.lookup id m)
fromListAF         = AF
                   . Map.fromListWith Set.union
                   . map (second Set.fromList)
toListAF (AF af)   = Map.toList (Map.map Set.toList af)
nullAF (AF af)     = Map.null af
unionAF (AF m1) (AF m2) =
    AF$ Map.unionWith Set.intersection m1 m2

concatAF []   = error "concatAF: cannot concat the empty set"
concatAF [af] = af
concatAF xx   = foldr1 union xx

mapAF :: (Identifier -> [Int] -> [Int]) -> AF -> AF
mapAF f (AF af) =
  AF$ Map.mapWithKey
      (\k ii -> Set.fromList (f k (Set.toList ii))) af

initAF t | sigt <- getSignature t = fromListAF
    [ (d, [0.. getArity sigt d -1])
          | d <-toList(definedSymbols sigt ‘mappend‘
                      constructorSymbols sigt)
          , getArity sigt d > 0]
```

```
instance Show AF where
  show = unlines
       . fmap show
       . fmap (second Set.toList)
       . Map.toList
       . fromAF
```

The most important operation is application of AFs to terms, rules or sets of rules. We introduce an overloaded operation `applyAF` by means of a type class. This is the traditional use of type classes to express adhoc polymorphism.

```
class ApplyAF t where applyAF :: AF -> t -> t
instance (Functor f, ApplyAF a) => ApplyAF (f a) where
    applyAF af = fmap (applyAF af)
instance (T Identifier :<: f) => ApplyAF (Term f) where
    applyAF af = foldTerm f
     where
          f t | Just (T (n::Identifier) tt) <- prj t
              , Just ii <- lookupAF n af = term n (select tt ii)
              | otherwise = inject t
instance ApplyAF (TRS Identifier f) where
    applyAF af trs@TRS{} = tRS$ applyAF af (rules trs)
```

We define now `afProcessor`, a monadic action in the `ProofF` monad. For every DP, we compute all the possible *minimal* AFs which make it ground by means of the function `findGroundAF`. The computed AFs are minimal in the sense that they never cut more information than needed. Next we sort and select the list of AFs according to some heuristic. In this case we use a simple definedness heuristic, selecting first those AFs which cut *less information*.

The result of `afProcessor` is an Or composition of a set of `Rewriting` problems, which can be now discharged to an external solver. We note that the definition of `afs` makes use of the `Set` restricted monad [Sittampalam and Gavin, 2008] to uniformly guarantee uniqueness of filtered positions.

```
afProcessor :: Problem f -> ProblemProof String f
afProcessor p@(Problem Narrowing trs dps@TRS{}) =
    if null orProblems
      then failP (AFProc Nothing) p "Could not find a grounding AF"
      else orP (AFProc Nothing) p orProblems
 where
  afs = findGroundAF p =<< Set.fromList (rules dps)
  orProblems =
      [ return $ applyAF af (Problem Rewriting trs dps)
          | af <- sortByDefinedness (Set.toList afs)
      ]
  sortByDefinedness = sortBy (flip compare `on` dpsSize)
  dpsSize af = sizeTRS (applyAF af dps)
  sizeTRS    = sum . fmap sum . fmap2 sizeTerm . rules
```

`on` is the well known combinator using for sorting a list on a view of the elements.

```
on cmp view x y = view x `cmp` view y
```

The battlehorse is the `Set`–monadic `findGroundAF` function, which must compute all the existing *sound* AFs which ground the rhs of the given DP. But first we must introduce a few intermediate tools. The condition of a sound AF is that the filtered TRS must not contain extra variables. We define an overloaded function `extraVars` to compute the extra variables of a rule or TRS.

```
class (IsVar f) => ExtraVars t f | t -> f
    where extraVars :: t -> [Term f]
instance (Ord (Term f), IsVar f) => ExtraVars (TRS id f) f where
    extraVars trs@TRS{} = concatMap extraVars (rules trs)
instance (Ord (Term f), IsVar f, Foldable f) => ExtraVars (Rule f) f
   where extraVars (l:->r) = nub (vars' r \\ vars' l)
```

`varsPositions` computes the list of variable positions in a term. For this we first annotate the subterms with their positions, then retrieve the variables in the term, and then just keep the positions and drop the terms.

```
varPositions :: (AnnotateWithPos f f, Var :<: f, Foldable f) =>
                 Term f -> [Position]
varPositions t = [ p | In(Note (p,t)) <- subterms (annotateWithPos t)
                     , Just Var{} <- [prj t] ]
```

The definition of findGroundAF, arguably one of the most involved pieces of code in Narradar, is given below. Roughly, we first compute all the *minimal* AFs which ground the rhs of the DP, and then just strengthen them until the *sound* invariant is fulfilled.

```
findGroundAF :: (Ord(Term f), AnnotateWithPos f f) =>
                Problem f -> DP f -> Set AF
findGroundAF p@(Problem _ trs@TRS{} dps) (_:->r)
  | isVar r   = mzero
  | otherwise = mkGround r >>= invariantEV
  where ...
```

mkGround returns the AF that cuts all the variables in a term t.

```
    mkGround :: Term f -> Set AF
    mkGround t = cutPP af0 t varsp
         where varsp   = varsPositions t
               af0     = initAF p
```

The cutPP helper computes all the *minimal* extensions to an AF af that cut a set of positions pp from a term t. It is defined in terms of cutP, which returns all the *minimal* extensions of an AF af to cut a position p from a term t (and fails for the root position which cannot be cut).

```
    cutPP :: AF -> Term f -> [Position] -> Set AF
    cutPP af t [] = return af
    cutPP af t pp = concatAF `liftM` (mapM (cutP af t) pp)

    cutP :: AF -> Term f -> Position -> Set AF
    cutP af t [] = mzero
```

```
cutP af t p  = Set.fromList
                [ cutAll [(root, [last sub_p])] af
                 | sub_p <- reverse (tail $ inits p)
                 , Just root <- [rootSymbol (t ! init sub_p)]]
```

The invariant is decomposed on two conditions: (1) that there are no extra vars in the rules, and (2) that there are no extra vars in the DPs. The fixpoint of the composition of these two subinvariants is just what we need.

```
invariantEV :: AF.AF -> Set AF.AF
invariantEV = fix (\f -> subinvariantEV trs f >=>
                         subinvariantEV dps f)
subinvariantEV :: TRS Identifier f ->
                  (AF -> Set AF) -> (AF -> Set AF)
subinvariantEV trs@TRS{} rec af
  | null extra = return af
  | otherwise  = foldM cutEV af (rules trs) >>= rec
        where extra = extraVars (applyAF af trs)
```

Finally `cutEV` simply returns the AF which cuts all the extra vars in a rule.

```
cutEV af rule@(_:->r)
  | extra <- extraVars (annotateWithPos <$> applyAF af rule)
  = cutPP af r (map note extra)
```

As said, `invariantEV` is the fixpoint of the composition of the two subinvariants, both defined by `subinvariantEV`, a function with explicit recursion knot `rec` which computes the extension of an AF to cut all the extra variables in a TRS, done via a Set-monadic fold over the list rules of the TRS. Moreover, note how in fact all the helper functions are monadic in `Set`. The use of the `Set` monad greatly simplifies the task of combining them in this problem.

4.2.4 The AProVe processor

The only remaining bit is to define a processor which calls AProVe with a rewriting problem and returs either a `Success` leaf, or a `Fail` leaf. Such a processor will necessarily make use of IO, and for this reason it must be defined on top of the `ProofT` monad transformer. We show only the type signature.

```
aproveProcessor :: Problem f -> ProblemProofT String IO f
```

4.2.5 Putting our solvers together

Composition of solvers in the `Proof` monad is simply monadic bind. However, while most of our solvers are in the `Proof` monad, we need to compose also the AProVe solver, which is in the `ProofT` monad over `IO`. In order to be able to compose the two types of solvers, we need to lift the pure ones to the `ProofT` monad. We do so by means of `wrap`.

```
liftProof :: Monad m => (a -> Proof f s b) -> a -> ProofT f s m b
liftProof f m = wrap (f m)
```

For instance, this is how we would compose the solvers described in this section for Narradar.

```
basicSolver :: Problem f -> ProblemProofT String IO f
basicSolver = liftProof cycleProcessor >=>
              liftProof afProcessor    >=>
                        aproveProcessor
```

Description of more complex strategies with ease is possible. Consider two hypothetical processors for forward instantiation and narrowing instantiation.

```
forwardProcessor, narrowingProcessor :: Problem f -> ProblemProof s f
```

We can define a strategy s.t. it first tries our basic solver above, and only if it fails then it tries to refine the graph using one of the processors below before trying again.

```
basic = liftProof afProcessor >=> aproveProcessor


strat = liftProof cycleProcessor >=>
        (basic .|. liftProof ((narrowingProcessor .|. forwardProcessor)
                              >=> cycleProcessor))
```

Above we make use of `MonadPlus` based alternative composition `(.|.)`. Note how we enforce the application of the dependency graph processor after any refinement of the graph.

```
(.|.) :: MonadPlus m => (b -> m a) -> (b -> m a) -> b -> m a
f .|. g = \x -> f x `mplus` g x
```

Running the processor is accomplished, essentially, by applying the solver to a concrete problem and then unwraping the `ProofT` to a `Proof` value.

```
runSolver problem solver = unwrap (solver problem)
```

4.2.6 Further points

- We have not described here how to avoid unnecessary work when running a proof search. Concretely, we regard as work the number of rewriting termination proofs requested to AProVe. The implementation described above fully explores the proof tree and apparently makes no effort to stop once a successful branch is found. But since Haskell is a lazy language and all what should be needed is careful control when manipulating (e.g. displaying) such a proof tree, see e.g. [Wadler, 1985] on the use lazyness for search. For instance, the `isSuccess` predicate we defined before will evaluate up to successful branches without requiring any changes. However, solvers defined in or lifted over the `IO` monad defeat lazyness, as `unwrap` will eagerly evaluate all the proof tree. Roughly, the solution to recover lazyness is either to define a custom interpreter in imperative style to replace `unwrap` by computing alternatives in `Or` branches sequentially and stopping as soon as success is found, or to carefully annotate `IO` processors with `unsafeInterleaveIO` to demand lazy IO.

- We also omitted the details of exploring different branches concurrently, which is crucial for a fast performing solver. Narradar makes a few efforts in this direction but we don't include the details here. Roughly, one can abstract over the shape of a `ProofT` computation and extract the list of all `Pure` leaves. The leaves can then be computed in parallel by the next processor and then reinserted in the shape; this works on any kind of proof tree and processor. But this is to naive, we don't really want to compute *all* the leaves in parallel; instead, we want to minimize speculative work by trying one path below every `Or` branch at a time.

- Narradar does several things with `ProofF` objects we have omitted in this report. There are modules to export a proof object as a HTML document, as a [Gra, N.d.] document for analysis of the proof search, or as a simple String for a command line interface. This could be extended to output proofs as XML, LaTeX, or as certificates for a theorem prover to verify the proof, as it is done in the CoLOR project[6].

4.3 Template Haskell derivations

```
instance Functor (ProofF f s)  where fmap    = fmapDefault
instance Foldable (ProofF f s) where foldMap = foldMapDefault
instance Foldable Problem_     where foldMap = foldMapDefault
instance Foldable RuleG        where foldMap = foldMapDefault
$(derive makeFunctor     ''Problem_)
$(derive makeTraversable ''Problem_)
-- $(derive makeFunctor     ''ProofF)    workaround for a TH problem
$(derive makeTraversable ''ProofF)
$(derive makeTraversable ''RuleG)
$(derive makeTraversable ''Var)
$(derive makeTraversable ''T)
```

[6]http://color.loria.fr

5 Conclusion

5.1 Related work

Although we have mentioned related work several times along the thesis, this section contains a more in-depth discussion of the state of the art.

The earliest positive result in the literature concerning the termination of ordinary narrowing was proved in [Christian, 1992] and holds for *left-flat* TRSs (each argument occurring at the lhs of a rewrite rule is either a variable — often called *shallow* [Comon, Haberstrau and Jouannaud, 1994]— or a ground term) compatible with a termination ordering. We can now give a better characterization of the left-flatness condition. Left-flatness ensures two key properties:

1. No defined symbols in the patterns of left hand sides mean no ll-dependency pairs.
2. No variables below the root position in left hand sides, together with compatibility with a termination ordering, means that no inductive constructions can be expressed. Therefore, such a TRS can always be translated to an equivalent non recursive encoding.

These two properties ensure the existence of a NDP termination proof for left–flat systems. To see why, note that due to point 1 we have no ll-dependency pairs, and due to point 2 we have no cycle between the lr-dependency pairs.

In recent work [Alpuente, Escobar and Iborra, 2008*b*], the authors classified a number of (mostly) syntactic restrictions under which narrowing is terminating. All the results in that paper, with the exception of Theorem 10[1], can be proved independently with the DPN method exposed in this paper. Briefly, and using the terminology introduced in that paper

- The srnf–based condition implies no ll-dependency pairs.
- The right–srnf condition implies no lr-dependency pairs.
- The rnf–based condition together with reachability completeness means no ll-dependency pairs can be involved in an infinite chain.
- The left–plain condition together with reachability completeness means no ll-dependency pairs can be involved in an infinite chain.
- The right–rnf condition together with reachability completeness means no lr-dependency pairs can be involved in an infinite chain.

From these points one can come up with suitable combinations to obtain the syntactic classes enumerated in [Alpuente, Escobar and Iborra, 2008*b*].

Two adaptations of the DP method to narrowing have recently appeared in the literature. [Nishida, Sakai and Sakabe, 2003] and [Nishida and Miura, 2006] introduced a similar, in power, method, which is restricted to classes of TRSs where narrowing has the TRAT property. We do away with this restriction and further develop their method with new, more obviously correct proofs, subsuming their method in all cases except for the case of TRSs with extra variables in the right hand side, which we do not consider.

The automatic method for the termination of narrowing with modes introduced in [Nishida and Vidal, 2008] has a component based on the DP method, but it also provides an alternative formulation based on the argument filtering transformation.

[1]and the result on linear goals, since in this work we do not parameterize on the starting goals

5.2 Future work

5.2.1 Restrictions and Strategies

We have not considered in this thesis restrictions of narrowing that improve its efficiency or termination properties, such as *Basic* Narrowing [Hullot, 1980*b*], *Needed* Narrowing [Antoy and Hanus, 1994], *Natural* Narrowing [**?**] or *Innermost* Narrowing [Bosco, Giovannetti and Moiso, 1988] to cite a few. Extending our method to handle these restricted forms of narrowing will surely enhance its usefulness and applicability, and we intend to pursue this goal at least for some of the enumerated restrictions. Particularly Innermost Narrowing which should be fairly direct, by taking advantage of all the work done on Q-restricted rewriting in modern formulations of the DP framework, e.g. see [Thiemann, 2007].

5.2.2 More Processors

We have shown how to adapt existing DP processors to the narrowing setting, and provided one processor to recast a narrowing DP problem as a rewriting DP problem. But there is still room for improvement, by adapting more rewriting processors or working on processors specific for narrowing.

Example 14 *Consider the TRS $\mathcal{R}$ formed by the single rule*

$$\mathtt{f}(\mathtt{a}(x), \mathtt{b}(y)) \to \mathtt{f}(y, y)$$

By analyzing it with our method we obtain a single narrowing dependency pair

$$\mathtt{f}^{\#}(\mathtt{a}(x), \mathtt{b}(y)) \to \mathtt{f}^{\#}(y, y)$$

There is only one choice of AF for making its right hand side ground, $\pi(f) = \{\}$, and we obtain the rewriting dependency pair

$$\mathtt{f}^{\#} \to \mathtt{f}^{\#}$$

which clearly constitutes a loop.

However $\mathcal{R}$ contains no infinite narrowing derivations.

5.3 Final Words

We have introduced a new technique for termination proofs of narrowing via termination of rewriting that is based on a suitable generalization of dependency pairs. Although several refinements of the notion of dependency pairs such as [Giesl et al., 2006; Hirokawa and Middeldorp, 2004] had been proposed previously for termination analysis of TRSs, this is the first time that the notion of dependency pair has been *extended* to deal with narrowing on arbitrary TRSs and queries. This is possible because we first identified the characterized the behaviour of infinite narrowing derivations with the notion of *echoing* terms.

Our contribution is threefold:

1. we ascertained the suitable notions that allow us to detect when the terms in a narrowing derivation actually do echo;
2. our approach leads to much weaker conditions for verifying the termination of narrowing that subsume all previously known termination of narrowing criteria;
3. the resulting method can be effectively mechanized. We have implemented our technique in a tool that is publicly available[2].

[2]http://safe-tools.dsic.upv.es/narradar

Bibliography

Alarcón, B., R. Gutiérrez, J. Iborra and S. Lucas. 2007. "Proving Termination of Context-Sensitive Rewriting with MU–TERM." *Electr. Notes Theor. Comput. Sci.* 188:105–115.

Alpuente, M., M. Falaschi and G. Levi. 1995. "Incremental Constraint Satisfaction for Equational Logic Programming." *Theoretical Computer Science* 142(1):27–57.

Alpuente, M., M. Falaschi and G. Vidal. 1994. Compositional Analysis for Equational Horn Programs. In *4th Int'l Conf. on Algebraic and Logic Programming, ALP'94.* Vol. 850 of *LNCS* Springer pp. 77–94.

Alpuente, M., M. Falaschi, M. Gabbrielli and G. Levi. 1993. The semantics of equational logic programming as an instance of CLP. In *Logic Programming Languages: Constraints, Functions and Objects*, ed. K. R. Apt, J. W. de Bakker and J. J. M. M. Rutten. Cambridge, Massachussets, USA: The MIT Press pp. 49–81.

Alpuente, M., S. Escobar and J. Iborra. 2008*a*. Modular Termination of Basic Narrowing. In *Proc. 19th Int'l Conf. on Rewriting Techniques and Applications, RTA'08.* LNCS Berlin: Springer-Verlag. To appear.

Alpuente, M., S. Escobar and J. Iborra. 2008*b*. "Termination of Narrowing revisited." *Theoretical Computer Science* . To appear.

Antoy, Sergio and Michael Hanus. 1994. A Needed Narrowing Strategy. In *Journal of the ACM.* ACM Press pp. 268–279.

Arts, T. and H. Zantema. 1996. Termination of Logic Programs Using Semantic Unification. In *Int'l Workshop on Logic-based Program Synthesis and Transformation.* Vol. 1048 of *Lecture Notes in Computer Science* Springer-Verlag pp. 219–233.

Arts, T. and J. Giesl. 2000. “Termination of Term Rewriting using Dependency Pairs.” *Theoretical Computer Science* 236(1-2):133–178.

Bosco, P. G., E. Giovannetti and C. Moiso. 1988. “Narrowing vs. SLD-resolution.” *Theoretical Computer Science* 59:3–23.

Christian, J. 1992. Some Termination Criteria for Narrowing and E-Narrowing. In *11th Int’l Conf. on Automated Deduction CADE’92.* Vol. 607 of *LNCS* Springer pp. 582–588.

Comon, Hubert, Marianne Haberstrau and Jean-Pierre Jouannaud. 1994. “Syntacticness, Cycle-Syntacticness, and Shallow Theories.” *Information and Computation* 111(1):154–191.

Dershowitz, N. 1995. Goal Solving as Operational Semantics. In *Int’l Logic Programming Symposium, ILPS’95.* Cambridge, MA: MIT Press pp. 3–17.

Dershowitz, N. 2003. Termination Dependencies. In *Proc. of the 6th Int’l Workshop on Termination.* Technical Report DSIC-II/15/03 pp. 27–30.

Erwig, Martin. 2001. “Inductive graphs and functional graph algorithms.” *J. Funct. Program.* 11(5):467–492.

Escobar, S., C. Meadows and J. Meseguer. 2006. “A Rewriting-Based Inference System for the NRL Protocol Analyzer and its Meta-Logical Properties.” *Theoretical Computer Science* 367(1-2):162–202.

Escobar, S. and J. Meseguer. 2007. Symbolic Model Checking of Infinite-State Systems Using Narrowing. In *18th Int’l Conference on Rewriting Techniques and Applications, RTA 2007.* Vol. 4533 of *Lecture Notes in Computer Science* Springer-Verlag pp. 153–168.

Fay, M. 1979. First-Order Unification in an Equational Theory. In *4th Int’l Conference on Automated Deduction, CADE’79.* pp. 161–167.

Giesl, J., R. Thiemann and P. Schneider-Kamp. 2005. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In

11th Int'l Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2005. Vol. 3452 of *LNCS* Springer pp. 301–331.

Giesl, J., R. Thiemann, P. Schneider-Kamp and S. Falke. 2004. Automated Termination Proofs with AProVE. In *Proc. 15th Int'l Conf. on Rewriting Techniques and Applications, RTA'04.* LNCS pp. 210–220.

Giesl, J., R. Thiemann, P. Schneider-Kamp and S. Falke. 2006. "Mechanizing and Improving Dependency Pairs." *J. Autom. Reasoning* 37(3):155–203.

Gra. N.d. "The GraphViz Project." http://www.graphviz.org.

Hanus, M. 1994. "The Integration of Functions into Logic Programming: From Theory to Practice." *Journal of Logic Programming* 19&20:583–628.

Hirokawa, N. and A. Middeldorp. 2004. Dependency Pairs Revisited. In *Proc. 15th Int'l Conf. on Rewriting Techniques and Applications, RTA'04.* Vol. 3091 of *LNCS* Springer pp. 249–268.

Hughes, John. 1995. The Design of a Pretty-printing Library. In *Advanced Functional Programming*, ed. Johan Jeuring and Erik Meijer. Vol. 925 of *Lecture Notes in Computer Science* Springer pp. 53–96.

Hullot, J.-M. 1980*a*. Canonical Forms and Unification. in *5th Int'l Conference on Automated Deduction CADE'80* [Hullot, 1980*b*] pp. 318–334.

Hullot, J.-M. 1980*b*. Canonical Forms and Unification. In *5th Int'l Conference on Automated Deduction CADE'80.* Vol. 87 of *LNCS* Berlin: Springer-Verlag pp. 318–334.

Kirchner, C., H. Kirchner and A. Santana de Oliveira. 2008. Analysis of Rewrite-Based Access Control Policies. In *Proc. 3rd Int'l Workshop on Security and Rewriting Techniques, SecreT 2008.* Elsevier ENTCS.

Mcbride, Conor and Ross Paterson. 2008. "Applicative programming with effects." *J. Funct. Program.* 18(1):1–13.

Meseguer, J. 1992. Multiparadigm logic programming. In *3rd Int'l Conference on Algebraic and Logic Programming, ALP'92.* Vol. 632 of *LNCS* Berlin: Springer-Verlag pp. 158–200.

Meseguer, J. and P. Thati. 2007. "Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols." *Higher-Order and Symbolic Computation* 20(1-2):123–160.

Nguyen, M. T., P. Schneider-Kamp, D. de Schreye and J. Giesl. 2008. Termination Analysis of Logic Programs based on Dependency Graphs. In *17th Int'l Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2007.* Vol. 4915 of *LNCS* Springer pp. 8–22.

Nishida, N. and G. Vidal. 2008. "Termination of Narrowing via Termination of Rewriting.". Submitted for Publication. Preliminary version in *Proc. FLOPS 2008*, LNCS 4989:113–129, 2008.

Nishida, N. and K. Miura. 2006. Dependency Graph Method for Proving Termination of Narrowing. In *8th Int'l Workshop on Termination, WST'06.* pp. 12–16.

Nishida, N., M. Sakai and T. Sakabe. 2003. "Narrowing-based simulation of term rewriting systems with extra variables." *Electr. Notes Theor. Comput. Sci.* 86(3).

Sheard, T. 2006. Type-Level Computation Using Narrowing in Ωmega. In *Programming Languages meets Program Verification.* Vol. 1643.

Sittampalam, Ganesh and Peter Gavin. 2008. "The Restricted Monad library." http://hackage.haskell.org/cgi-bin/hackage-scripts/package/rmonad.

Swierstra, Wouter. 2008. "Data types la carte." *Journal of Functional Programming* 18(04):423–436.

TeReSe, ed. 2003. *Term Rewriting Systems.* Cambridge, UK: Cambridge University Press.

Thiemann, René. 2007. The DP Framework for Proving Termination of Term Rewriting PhD thesis RWTH Aachen University.

Wadler, Philip. 1985. How to replace failure by a list of successes. In *Proc. of a conference on Functional programming languages and computer architecture.* New York, NY, USA: Springer-Verlag New York, Inc. pp. 113–128.