Higher Order Dependency Pairs for Algebraic Functional Systems

Cynthia Kop[1] **and Femke van Raamsdonk**[1]

1 **Faculty of Sciences, VU, De Boelelaan 1081a, 1081 HV Amsterdam**

Abstract

We extend the termination method using dynamic dependency pairs to higher order rewriting systems with beta as a rewrite step, also called Algebraic Functional Systems (AFSs). We introduce a variation of usable rules, and use monotone algebras to solve the constraints generated by dependency pairs. This approach differs in several respects from those dealing with higher order rewriting modulo beta (e.g. HRSs).

Keywords and phrases higher order rewriting, termination, dynamic dependency pairs

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.203

Category Regular Research Paper

1 Introduction

An important method to (automatically) prove termination of first order term rewriting is the dependency pair approach by Arts and Giesl [3]. This approach transforms a rewrite system into groups of ordering constraints, such that rewriting is terminating if and only if the groups of constraints are (separately) solvable. Various optimizations of the method have been studied, see for example [7, 6].

This paper contributes to the study of dependency pairs for higher order rewriting. Higher order rewriting comes in different shapes. First, there is rewriting modulo $\alpha\beta\eta$ as in the higher order rewrite systems (HRSs) defined by Nipkow [20]; Klop's CRSs [13] and Khasidashvili's ERSs [12] are in some aspects similar. Various definitions of dependency pairs, often with optimizations, have been given for HRSs [23, 22, 17, 15, 24]. Second, applicative term rewriting systems with functional variables but no abstraction are sometimes considered as a (restricted) form of higher order rewriting. Also in this setting several definitions of dependency pairs exist [16, 18, 19, 1, 2, 8]. The aim of the present paper is to study dependency pairs for a third variant of higher order rewriting: *algebraic functional systems* (AFSs), introduced by Jouannaud and Okada [10]. In AFSs we consider simply typed terms, which are rewritten both using specific rewrite rules and β-reduction, with matching modulo α. While higher order versions of the recursive path ordering are commonly studied in the setting of AFSs [11, 5], there is little work on dependency pairs for this formalism.

We briefly discuss the ideas from studies of dependency pairs for HRSs and for applicative systems in Section 2; we also explain why those approaches do not quite, or not at all apply to the setting with AFSs. We define dependency pairs for AFSs in the so-called *dynamic* style, where functional variables in the right-hand side of a rewrite rule may give rise to dependency pairs. We study the notions of dependency chains, dependency graphs and reduction orders for AFSs with dynamic dependency pairs. To demonstrate that the dynamic approach has adequate strength even without restrictions, we also define a variant of usable rules and apply van de Pol's monotone algebra approach [21] to solve constraints generated by the method. The result is a method to prove termination (a complete method for left-linear systems), which may serve as a basis for further definitions – for example static dependency pairs, or dynamic pairs with restrictions that allow us to drop the subterm property.

2 Background and Related Work

The extension of dependency pairs to the higher order case is not entirely straightforward and thus many variations exist. This work can roughly be split along two axes. On the one axis, the higher order formalism (we distinguish between applicative rewriting, rewriting modulo β (HRSs), and with β as a separate step (AFSs)), on the other the style of dependency pairs (with the common styles being *dynamic* and *static*). Figure 1 gives an overview.

	Applicative	**HRS**	**AFS**
Dynamic	[16]	[23] [15]	*this paper*
Static	[18] [19]	[4] [22] [17] [24]	[4]
Other	[1] [2] [8]	–	–

Figure 1 References on Higher Order Dependency Pairs

The dynamic and static approach differ in the treatment of leading variables in the right-hand sides of rules (subterms $x \cdot s_1 \cdots s_n$ with $n > 0$ and x a free variable). In the dynamic approach, such subterms lead to a dependency pair; in the static approach they do not. Consequently, first order techniques like argument filterings and usable rules are easier to extend to a static approach, while equivalence results tend to be limited to the dynamic style. Static dependency pairs can only be applied on systems satisfying certain restrictions.

Dependency pairs for applicative term rewriting We first say some words about applicative term rewriting. In applicative systems, terms are built from variables, constants and a binary application operator. Functional variables may be present, as in $x \cdot a$, but there is no abstraction, as in $\lambda x. x$. There are various styles of applicative rewriting.

A dynamic approach was defined both for untyped and simply-typed applicative systems in [16], along with a definition of argument filterings. A first static approach appears in [18] and is improved in [19]; the method is restricted to 'plain function passing' systems where, intuitively, leading variables are harmless. Due to the lack of binders, it is also possible to eliminate leading variables by instantiating them, as is done for simply typed systems in [1, 2]; in [8], an uncurrying transformation from untyped applicative systems to normal first order systems is used. These techniques have no parallel in rewriting with binders.

Unfortunately, they are not directly useful in the setting of AFSs, since termination may be lost by adding λ-abstraction and β-reduction. For example, the simply typed applicative system $\mathsf{app} \cdot (\mathsf{abs} \cdot F) \cdot x \to F \cdot x$, with $F : \iota \Rightarrow \iota$ a functional variable, $x : \iota$ a variable, and app, abs constants, is terminating because in every step the size of a term decreases. However, adding λ-abstraction and β-reduction spoils this property: with $\omega = \mathsf{abs} \cdot (\lambda x. \mathsf{app} \cdot x \cdot x)$ we have $\mathsf{app} \cdot \omega \cdot \omega = \mathsf{app} \cdot (\mathsf{abs} \cdot (\lambda x. \mathsf{app} \cdot x \cdot x)) \cdot \omega \to (\lambda x. \mathsf{app} \cdot x \cdot x) \cdot \omega \to \mathsf{app} \cdot \omega \cdot \omega$.

Dynamic Dependency Pairs for HRSs A first, very natural, definition of dependency pairs for HRSs is given in [23]. Here termination is not equivalent to the absence of infinite dependency chains, and a term is required to be greater than its subterms (the *subterm property*), which makes many optimizations impossible. Consequently, most of the focus since has been on the static approach. However, with restrictions on the rules the subterm property may be weakened, as discussed in [15] (extended abstract).

Static Dependency Pairs for HRSs The static approach in [18] is moved to the setting of HRSs in [17], and extended with argument filterings and usable rules in [24]. The static approach omits dependency pairs $f^{\#}(\vec{l}) \rightsquigarrow x(\vec{r})$ with x a variable, which avoids the need of a subterm property. The technique is restricted to *plain function passing* HRSs; for example the (terminating) rule $\mathsf{foo}(\mathsf{bar}(\lambda x. F(x))) \to F(\mathsf{a})$ cannot be handled. In addition, bound variables may become free in a dependency pair. For instance, the rule $\mathsf{I}(\mathsf{s}(n)) \to \mathsf{twice}(\lambda x. I(x), n)$ generates a pair $\mathsf{I}^{\#}(\mathsf{s}(n)) \rightsquigarrow \mathsf{I}^{\#}(x)$ which admits an infinite dependency chain.

The definitions for HRSs [23, 17] do not immediately carry over to AFSs, since AFSs may have rules of functional type and β-reduction is a separate rewrite step. A short paper by Blanqui [4] introduces static dependency pairs on a form of rewriting which includes AFSs, but it restricts to base-type rules. The present work considers dynamic dependency pairs and is most related to [23], but is adaptated for the different formalism. Our method conservatively extends the one for first order rewriting and provides a characterization of termination for left-linear AFSs. We have chosen for a dynamic rather than a static approach because, although the static approach is stronger when applicable, the dynamic definitions can be given without restrictions. It would be nice for future work to integrate the two approaches; for the moment they co-exist with each their own advantages and disadvantages.

3 Preliminaries

We consider higher order rewriting as defined by Jouannaud and Okada, also called Algebraic Functional Systems (AFSs). Terms are built from simply typed variables, abstraction and application (as in simply typed λ-calculus), and in addition function symbols which take a fixed number of typed arguments. Terms and matching are modulo α, and β is a rewrite step. We follow roughly the definitions in [25, Chapter 11], as recalled below.

Types and Terms The set of *simple types* (or just *types*) is generated from a given set $\mathcal{B}$ of *base types* and the binary type constructor $\Rightarrow$, which is right-associative. Types are denoted by σ, τ and base types by ι, κ. A type with at least one occurrence of $\Rightarrow$ is called a *functional type*. A *type declaration* is an expression of the form $(\sigma_1 \times \ldots \times \sigma_n) \Rightarrow \tau$; if $n = 0$ this is written as just τ. Type declarations are not types, but are used for typing purposes.

We assume a set $\mathcal{V}$, consisting of infinitely many typed variables for each type, and a set $\mathcal{F}$ disjoint from $\mathcal{V}$, consisting of function symbols each equipped with a type declaration. Variables are denoted by x, y, z and function symbols by f, g, h or using more suggestive notation. To stress the type (declaration) of a symbol a we may write $a : \sigma$. *Terms* over $\mathcal{F}$ are those expressions s for which we can infer $s : \sigma$ for some type σ using the clauses:

`(var)`	$x : \sigma$	if $x : \sigma \in \mathcal{V}$
`(app)`	$s \cdot t : \tau$	if $s : \sigma \Rightarrow \tau$ and $t : \sigma$
`(abs)`	$\lambda x. s : \sigma \Rightarrow \tau$	if $x : \sigma \in \mathcal{V}$ and $s : \tau$
`(fun)`	$f(s_1, \ldots, s_n) : \tau$	if $f : (\sigma_1 \times \ldots \times \sigma_n) \Rightarrow \tau \in \mathcal{F}$ and $s_1 : \sigma_1, \ldots, s_n : \sigma_n$

Note that a function symbol $f : (\sigma_1 \times \ldots \times \sigma_n) \Rightarrow \tau$ takes exactly n arguments, and τ is not necessarily a base type. λ binds occurrences of variables as in the λ-calculus. Terms are considered modulo α-conversion; bound variables are renamed if necessary. The set of variables of s which are not bound is denoted $FV(s)$. Application is left-associative.

A *substitution* $[\vec{x} := \vec{s}]$, with $\vec{x}$ and $\vec{s}$ non-empty finite vectors of equal length, is the homomorphic extension of the type-preserving mapping $\vec{x} \mapsto \vec{s}$ from variables to terms. Substitutions are denoted γ, δ, and the result of applying γ to a term s is denoted $s\gamma$. The *domain* $\mathsf{dom}(\gamma)$ of $\gamma = [\vec{x} := \vec{s}]$ is $\{\vec{x}\}$. Substituting does not capture free variables.

We assume a fresh symbol $\Box_\sigma : \sigma$ for every type σ. A *context* $C[]$ is a term with a single occurrence of some $\Box_\sigma$. The result of replacing $\Box_\sigma$ in $C[]$ by a term s of type σ is denoted

$C[s]$. Free variables may be captured; if $C[] = \lambda x. \square_\sigma$ then $C[x] = \lambda x. x$. If $s = C[t]$ we say t is a *subterm* of s, notation $s \trianglerighteq t$, or $s \vartriangleright t$ (strict subterm) if $C[]$ is not the empty context $\square$.

Rules and Rewriting A *rewrite rule* is a pair of terms $l \to r$ such that l and r are terms of the same type and do not contain a subterm of the form $(\lambda x. s) \cdot t$, all free variables of r also occur in l, and l has the form $f(l_1, \dots, l_n) \cdot l_{n+1} \cdots l_m$ (with $m \geq n \geq 0$). Given a set of rules $\mathcal{R}$, the *rewrite* or *reduction relation* $\to_{\mathcal{R}}$ on terms is given by the following clauses:

`(rule)`	$C[l\gamma]$	$\to_{\mathcal{R}}$	$C[r\gamma]$	with $l \to r \in \mathcal{R}$, C a context, γ a substitution
`(beta)`	$C[(\lambda x. s) \cdot t]$	$\to_{\mathcal{R}}$	$C[s[x := t]]$	

We sometimes use the notation $s \to_\beta t$ for a rewrite step using `(beta)`. An *algebraic functional system* (AFS) is the combination of a set of terms and a rewrite relation on this set, and is usually specified by a set of rules (perhaps with function symbols). A function symbol f is a *defined symbol* of an AFS if there is a rule with left-hand side $f(l_1, \dots, l_n) \cdot l_{n+1} \cdots l_m$. A function symbol that is not a defined symbol is a *constructor symbol*. The sets of defined or constructor symbols are denoted by $\mathcal{D}$ or $\mathcal{C}$ respectively. A rewrite rule $l \to r$ is *left-linear* if every free variable occurs at most once in l; an AFS is left-linear if all its rewrite rules are.

▶ **Example 3.1.** Throughout this paper, we will consider as an example the AFS twice. It has four function symbols, $\mathsf{o} : \mathsf{nat}$, $\mathsf{s} : (\mathsf{nat}) \Rightarrow \mathsf{nat}$, $\mathsf{I} : (\mathsf{nat}) \Rightarrow \mathsf{nat}$, $\mathsf{twice} : (\mathsf{nat} \Rightarrow \mathsf{nat}) \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}$, and three rewrite rules:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathsf{I}(\mathsf{o}) & \to & \mathsf{o} & \mathsf{twice}(F) & \to & \lambda y. F \cdot (F \cdot y) \\ \mathsf{I}(\mathsf{s}(n)) & \to & \mathsf{s}(\mathsf{twice}(\lambda x. \mathsf{I}(x)) \cdot n) & & & \end{array}$$

An example reduction: $\mathsf{I}(\mathsf{s}(\mathsf{o})) \to \mathsf{s}(\mathsf{twice}(\lambda x. \mathsf{I}(x)) \cdot \mathsf{o}) \to \mathsf{s}((\lambda y. (\lambda x. \mathsf{I}(x)) \cdot ((\lambda x. \mathsf{I}(x)) \cdot y)) \cdot \mathsf{o}) \to_\beta \mathsf{s}((\lambda x. \mathsf{I}(x)) \cdot ((\lambda x. \mathsf{I}(x)) \cdot \mathsf{o})) \to_\beta \mathsf{s}((\lambda x. \mathsf{I}(x)) \cdot \mathsf{I}(\mathsf{o})) \to \mathsf{s}((\lambda x. \mathsf{I}(x)) \cdot \mathsf{o}) \to_\beta \mathsf{s}(\mathsf{I}(\mathsf{o})) \to \mathsf{s}(\mathsf{o})$.

The symbol I represents the identity function, and therefore no infinite reduction exists. However, this is not trivial to prove; neither orderings like HORPO [11] nor a static dependency pair approach can handle the second rule, due to the subterm $\mathsf{I}(x)$. The static approach gives a requirement $\mathsf{I}^\#(\mathsf{s}(n)) > \mathsf{I}^\#(x)$, where the right-hand side contains a variable which does not occur in the left-hand side. Since $>$ must be closed under substitution, this is impossible to satisfy, as $\mathsf{s}(n)$ might be substituted for x. Applying HORPO leads to a similar problem.

4 Dependency Pairs

An intuition behind the dependency pair approach is to identify those parts of the right-hand sides of rewrite rules which may give rise to an infinite reduction. These are subterms headed by a defined symbol (as in first order term rewriting), and also subterms headed by a free variable, because such a variable can be instantiated by a defined symbol or abstraction. The latter is typical for the *dynamic* approach to higher order dependency pairs.

In this section we will extend the concepts of dependency pairs and dependency chains to AFSs. We show that an AFS is terminating if it does not have an infinite dependency chain, and that absence of dependency chains characterizes termination for left-linear AFSs.

Completed Rules An AFS is *completed* by adding for each rule of the form $l \to \lambda x_1 \dots x_n. r$ with $n > 0$ and r not an abstraction the n new rules $l \cdot x_1 \to \lambda x_2 \dots x_n. r, \dots, l \cdot x_1 \cdot \dots \cdot x_n \to r$. We do this to avoid creating dependency pairs containing a β-redex. Completing does not affect termination. For example, the system twice is completed by adding $\mathsf{twice}(F) \cdot m \to F \cdot (F \cdot m)$. In the remainder of the paper, we work with completed AFSs.

Candidate terms The definition of a dependency pair uses the notion of candidate terms, intuitively those subterms which might cause non-termination. Subterms that cannot be reduced at the root are omitted, because they are not a minimal starting point of an infinite reduction. Bound variables that become free by taking a subterm are replaced by fresh constants. We denote by $\mathbb{C}$ the set consisting of infinitely many fresh symbols c_x with c_x the same type as x, where x in c_x is not bound and is not subject to α-conversion.

▶ **Definition 4.1.** We say $t[x_1 := c_{x_1}, \dots, x_n := c_{x_n}]$ is a *candidate term* of s if $s \trianglerighteq t$, and $x_1, \dots, x_n$ are the variables which occur bound in s but free in t, and either $t = f(t_1, \dots, t_n) \cdot t_{n+1} \cdots t_m$ with f a defined symbol and $m \geq n \geq 0$, or $t = x \cdot t_1 \cdots t_n$ with x free in s and $n > 0$. We denote the set of candidate terms of s by $Cand(s)$.

In the AFS twice we have $Cand(F \cdot (F \cdot m)) = \{F \cdot (F \cdot m),\ F \cdot m\}$ and $Cand(\mathsf{s}(\mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot n)) = \{\mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot n, \mathsf{twice}(\lambda x.\, \mathsf{I}(x)),\ \mathsf{I}(c_x)\}$. Note that for example $x \cdot y$ is not a candidate term of $g(\lambda x.\, x \cdot y)$ because x occurs only bound.

Dependency Pairs The definition of dependency pair also uses marked function symbols as in the first order case. Let $\mathcal{F}^\# = \mathcal{F} \cup \{f^\# : \sigma \mid f : \sigma \in \mathcal{D}\}$, so $\mathcal{F}$ extended with a marked version for every defined symbol, having the same type declaration. The marked counterpart of a term s, notation $s^\#$, is $f^\#(s_1, \dots, s_n)$ if $s = f(s_1, \dots, s_n)$ with f in $\mathcal{D}$, and just s otherwise. For example, $(\mathsf{twice}(F))^\# = \mathsf{twice}^\#(F)$ and $(\mathsf{twice}(F) \cdot m)^\# = \mathsf{twice}(F) \cdot m$.

▶ **Definition 4.2** (Dependency Pair)**.** The set of *dependency pairs* of a rewrite rule $l \to r$, notation $\mathsf{DP}(l \to r)$, consists of:

- all pairs $l^\# \rightsquigarrow p^\#$ with $p \in Cand(r)$,
- all pairs $l \cdot y_1 \cdots y_k \rightsquigarrow r \cdot y_1 \cdots y_k$ with $1 \leq k \leq n$ if $r : \sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \iota$, and either $r = x \cdot r_1 \cdots r_m$ with $m \geq 0$ or $r = f(r_1, \dots, r_i) \cdot r_{i+1} \cdots r_m$ with $m \geq i \geq 0$ and $f \in \mathcal{D}$.

We use $\mathsf{DP}(\mathcal{R})$ (or just DP) for the set of all dependency pairs of rewrite rules of an AFS $\mathcal{R}$.

▶ **Example 4.3.** The set of dependency pairs of the AFS twice consists of:

$$\begin{array}{lcllcl}
\mathsf{I}^\#(\mathsf{s}(n)) & \rightsquigarrow & \mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot n & \mathsf{twice}^\#(F) & \rightsquigarrow & F \cdot (F \cdot c_y) \\
\mathsf{I}^\#(\mathsf{s}(n)) & \rightsquigarrow & \mathsf{twice}^\#(\lambda x.\, \mathsf{I}(x)) & \mathsf{twice}^\#(F) & \rightsquigarrow & F \cdot c_y \\
\mathsf{I}^\#(\mathsf{s}(n)) & \rightsquigarrow & \mathsf{I}^\#(c_x) & \mathsf{twice}(F) \cdot m & \rightsquigarrow & F \cdot (F \cdot m) \\
 & & & \mathsf{twice}(F) \cdot m & \rightsquigarrow & F \cdot m
\end{array}$$

The last two dependency pairs originate from the rule added by completion.

To illustrate the second form of dependency pair, consider the system with function symbols $\mathsf{app} : (o) \Rightarrow o \Rightarrow o$ and $\mathsf{abs} : (o \Rightarrow o) \Rightarrow o$, and one rewrite rule: $\mathsf{app}(\mathsf{abs}(x)) \to x$. This system has no dependency pairs of the first form, but does admit a two-step loop: $s := \mathsf{app}(\mathsf{abs}(\lambda x.\, \mathsf{app}(x) \cdot x)) \cdot \mathsf{abs}(\lambda x.\, \mathsf{app}(x) \cdot x) \to (\lambda x.\, \mathsf{app}(x) \cdot x) \cdot \mathsf{abs}(\lambda x.\, \mathsf{app}(x) \cdot x) \to_\beta s$.

Comparing our approach to static dependency pairs as defined in [17], the two main differences are that we avoid bound variables becoming free, and that we include dependency pairs where the right-hand side is headed by a variable. We call such pairs *collapsing*.

Dependency Chains We can now investigate termination by means of *dependency chains*:

▶ **Definition 4.4.** A *dependency chain* is a sequence $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ such that for all i:

1. $\rho_i \in \mathsf{DP} \cup \{\mathtt{beta}\}$,
2. if $\rho_i = l_i \rightsquigarrow p_i \in \mathsf{DP}$ there exists γ with domain $FV(l_i)$ such that $s_i = l_i\gamma$ and $t_i = p_i\gamma$

3. if $\rho_i = \texttt{beta}$ then $s_i = (\lambda x. u) \cdot v \cdot w_1 \cdots w_k$ and either
   a. $k > 0$ and $t_i = u[x := v] \cdot w_1 \cdots w_k$, or
   b. $k = 0$ and there is w such that $u \trianglerighteq w$ and $x \in FV(w)$ and $w^{\#}[x := v] = t_i$, but $w \neq x$
4. $t_i \rightarrow_{in}^{*} s_{i+1}$

A step $\rightarrow_{in}$ is obtained by rewriting some s_i inside a term of the form $f(s_1, \dots, s_n) \cdot s_{n+1} \cdots s_m$.

▶ **Theorem 4.5.** *If $\mathcal{R}$ is non-terminating there is an infinite dependency chain over* $\mathsf{DP}(\mathcal{R})$.

Proof Sketch. Say a term s is *minimally non-terminating* (MNT) if s is terminating but all its subterms are not. Let u_{-1} be any MNT term, and subsequently for every $i \in \mathbb{N}$, given an MNT term u_{i-1}, we define $\rho_i \in \mathsf{DP} \cup \{\texttt{beta}\}$ and terms s_i and t_i. Note (**): *if an MNT term is reduced at any other position than the top, the result is also MNT, or terminating.*

If $u_{i-1} = (\lambda x. s) \cdot t$ then $s[x := t]$ is also non-terminating (because eventually a topmost step must be done, and we can see that $s[x := t]$ reduces to the result); let u_i be an MNT subterm of $s[x := t]$ and define $\rho_i, s_i, t_i := \texttt{beta}, u_{i-1}, u_i^{\#}$. If $u_{i-1} = (\lambda x. s) \cdot t \cdot v_0 \cdots v_k$ then by (**) $u_i := s[x := t] \cdot v_0 \cdots v_k$ is also MNT, so choose $\rho_i, s_i, t_i := \texttt{beta}, u_{i-1}, u_i$. Otherwise $u_{i-1} = f(v_1, \dots, v_n) \cdot v_{n+1} \cdots v_m$; then $u_{i-1} \rightarrow_{in}^{*}$ some term $l\gamma \cdot w_1 \cdots w_k$, with $r\gamma \cdot \vec{w}$ still non-terminating. If $k > 0$ then by (**) $r\gamma \cdot \vec{w}$ is MNT, so choose $u_i := r\gamma \cdot \vec{w}$ and $\rho_i, s_i, t_i := l \cdot x_1 \cdots x_k \rightsquigarrow r \cdot x_1 \cdots x_k,\ l\gamma \cdot \vec{w}, r\gamma \cdot \vec{w}$. Otherwise let r' be the smallest subterm of r such that $p := r'\delta$ is still non-terminating, where δ replaces the newly free variables x_i by c_{x_i}. Then some analysis shows that p is a candidate of r and $p\gamma$ is also MNT; choose $u_i := p\gamma$ and $\rho_i, s_i, t_i := l^{\#} \rightsquigarrow p^{\#}, l^{\#}\gamma, p^{\#}\gamma$. This process generates a dependency chain. ◀

The converse of Theorem 4.5 does not hold. Consider the AFS with rules:

$$f(x, y, \mathsf{s}(z)) \rightarrow g(h(x, y), \lambda u. f(u, x, z)) \quad \text{and} \quad h(x, x) \rightarrow f(x, \mathsf{s}(x), \mathsf{s}(\mathsf{s}(x)))$$

This system has the following dependency pairs:

$$\begin{array}{lclclcl} f^{\#}(x, y, \mathsf{s}(z)) & \rightsquigarrow & h^{\#}(x, y) & \quad & h^{\#}(x, x) & \rightsquigarrow & f^{\#}(x, \mathsf{s}(x), \mathsf{s}(\mathsf{s}(x))) \\ f^{\#}(x, y, \mathsf{s}(z)) & \rightsquigarrow & f^{\#}(c_u, x, z) & & & & \end{array}$$

There is an infinite dependency chain: $f^{\#}(c_u, \mathsf{s}(c_u), \mathsf{s}(\mathsf{s}(c_u))) \rightsquigarrow f^{\#}(c_u, c_u, \mathsf{s}(c_u)) \rightsquigarrow h^{\#}(c_u, c_u)$ $\rightsquigarrow f^{\#}(c_u, \mathsf{s}(c_u), \mathsf{s}(\mathsf{s}(c_u))) \rightsquigarrow \dots$ However, the AFS is terminating, intuitively because the bound variable destroys matching possibilities. The crucial point of the example is the combination of bound variables and non-left-linear rules. Theorem 4.6 shows that for left-linear AFSs, the absence of infinite dependency chains actually characterizes termination.

▶ **Theorem 4.6.** *A left-linear AFS $\mathcal{R}$ is terminating if and only if it does not admit an infinite dependency chain.*

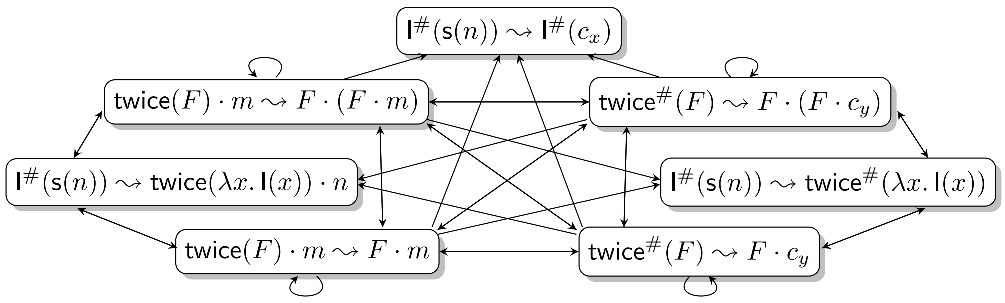
Proof Sketch. In a left-linear system replacing variables by a function symbol that doesn't occur in any rule has no effect on applicability of $\rightarrow_{\mathcal{R}}$. Thus a dependency chain effectively produces an infinite reduction $|s_i| \rightarrow_{\mathcal{R}} \cdot \rhd |t_i| \rightarrow_{\mathcal{R}}^{*} |s_{i+1}|$ (where $|\cdot|$ replaces any $f^{\#}$ by its unmarked counterpart), and this implies the existence of an infinite $\rightarrow_{\mathcal{R}}$ reduction. ◀

5 The Dependency Graph

As in the first order case, we use a dependency graph to organize the dependency pairs. The definition of a dependency graph is typical for our setting here, namely AFSs with dynamic dependency pairs, but the other notions we use are similar to the first order ones.

The *dependency graph* of an AFS $\mathcal{R}$ has the dependency pairs of $\mathcal{R}$ as nodes, and an edge from node $l \leadsto p$ to node $l' \leadsto p'$ if there is a finite dependency chain $[(l \leadsto p, s_1, t_1), (\texttt{beta}, s_2, t_2), \ldots, (\texttt{beta}, s_{k-1}, t_{k-1}), (l' \leadsto p', s_k, t_k)]$ with all but the first and the last elements $\texttt{beta}$.

▶ **Example 5.1.** The dependency graph of the AFS twice:



$\mathsf{I}^\#(\mathsf{s}(n)) \leadsto \mathsf{I}^\#(c_x)$
$\mathsf{twice}(F) \cdot m \leadsto F \cdot (F \cdot m)$
$\mathsf{twice}^\#(F) \leadsto F \cdot (F \cdot c_y)$
$\mathsf{I}^\#(\mathsf{s}(n)) \leadsto \mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot n$
$\mathsf{I}^\#(\mathsf{s}(n)) \leadsto \mathsf{twice}^\#(\lambda x.\, \mathsf{I}(x))$
$\mathsf{twice}(F) \cdot m \leadsto F \cdot m$
$\mathsf{twice}^\#(F) \leadsto F \cdot c_y$

A *cycle* is a non-empty set $\mathcal{C}$ of dependency pairs such that between every two pairs $\rho, \pi \in \mathcal{C}$ there is a non-empty path in the graph using only nodes in $\mathcal{C}$. A cycle that is not contained in any other cycle is called a *strongly connected component* (SCC). To prove termination we must show that cycles in a dependency graph are in some sense well-behaved (see Theorem 6.2). Due to clause 3b in Definition 4.4, there is an edge from any node of the form $l \leadsto x \cdot r_1 \ldots \cdot r_n$ with x a variable to all other nodes. Hence a rule with a functional variable in its right-hand side gives rise to many cycles. Here, exactly, lies the appeal of the static approach, which eliminates the need for such pairs. However, this barrier is not impossible to overcome, and as discussed, the dynamic approach can deal with systems where the static approach fails.

A set $D \subseteq \mathsf{DP}$ is *looping* if there is an infinite dependency chain using only dependency pairs from D and $\texttt{beta}$. By termination of simply typed β-reduction, $\emptyset$ is not looping.

Because the dependency graph cannot be computed in general, one uses *approximations* of the dependency graph, which have the same nodes but possibly more edges. A brute method to find an approximation of the dependency graph is to have an edge between $l \leadsto p$ and $l' \leadsto p'$ as soon as the head of p is a variable, or if p and l' both have the form $f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m$ for some function symbol f and some $m \geq n \geq 0$. It is interesting to study more sophisticated methods to find approximations, but this is left for future work.

In the remainder of this paper, we will assume that dependency graphs (and hence also their approximations) have only finitely many nodes. This is the case if the AFS under consideration has finitely many rewrite rules. However, note that also for infinite AFSs (arising for example by instantiation of polymorphic rewrite rules) we can work with finite dependency graphs, if (infinite) sets of dependency pairs are represented by a single node.

▶ **Lemma 5.2.** *Let G be an approximation of the dependency graph of an AFS $\mathcal{R}$. Suppose that every cycle in G is non-looping. Then $\mathcal{R}$ is terminating.*

Proof Sketch. Given an infinite dependency chain, there must be a dependency pair ρ_i which occurs infinitely often (by the finiteness assumption). Then $\{\rho_j \mid j > i\}$ is a cycle. ◀

▶ **Example 5.3.** The dependency graph (approximation) of twice from Example 5.1 admits many cycles, such as $\{\mathsf{twice}(F) \cdot n \leadsto F \cdot (F \cdot n)\}$ or the following cycle $\mathcal{C}_{\mathsf{twice}}$:

$$\left\{ \begin{array}{rclrcl} \mathsf{I}^\#(\mathsf{s}(n)) & \leadsto & \mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot n & \mathsf{twice}^\#(F) & \leadsto & F \cdot (F \cdot c_y) \\ \mathsf{I}^\#(\mathsf{s}(n)) & \leadsto & \mathsf{twice}^\#(\lambda x.\, \mathsf{I}(x)) & \mathsf{twice}^\#(F) & \leadsto & F \cdot c_y \\ \mathsf{twice}(F) \cdot m & \leadsto & F \cdot (F \cdot m) & \mathsf{twice}(F) \cdot m & \leadsto & F \cdot m \end{array} \right\}$$

$\mathcal{C}_{\mathsf{twice}}$ is an SCC and includes all cycles. Therefore twice is terminating if $\mathcal{C}_{\mathsf{twice}}$ is non-looping.

6 Reduction Orders

The challenge, then, is to prove the absence of looping cycles. We use the following definition:

▶ **Definition 6.1.** A *reduction triple* consists of a well-founded ordering $>$, a quasi-ordering $\geq$ and a sub-relation $\geq_1$ of $\geq$, such that:

1. $>$ and $\geq$ are *compatible*: either $> \cdot \geq \subseteq >$ or $\geq \cdot > \subseteq >$;
2. $>$, $\geq$ and $\geq_1$ are all *stable* (that is, closed under substitution);
3. $\geq_1$ is *monotonic*: (that is, if $s \geq_1 t$ with s, t sharing a type, then $C[s] \geq_1 C[t]$);
4. $\geq_1$ contains `beta` (that is, always $(\lambda x.\, s) \cdot t \geq_1 s[x := t]$).

A *reduction pair* is a pair $(>, \geq)$ such that $(>, \geq, \geq)$ is a reduction triple; this corresponds with the original (first order) notion of reduction pair. The reduction triple is a generalisation of this notion, where $\geq$ itself is not required to be monotonic; we will need a non-monotonic $\geq$ in Section 6.1 to compare terms with different types. To deal with subterm reduction in dependency chains, an additional definition is needed. We say $\geq$ has the *limited subterm property* if: for all x, s, t, u such that $s \trianglerighteq u$ and u is neither an abstraction nor a single variable, there is a substitution γ such that $(\lambda x.\, s) \cdot t \geq (u^{\#})\gamma[x := t]$. Intuitively, the substitution γ is used to replace free variables in u that are bound in s by fresh constants c_x. However, we will also use a more liberal replacement of those variables, hence the general γ.

The following theorem shows how reduction triples can be used with dependency pairs.

▶ **Theorem 6.2.** *A set $D = D_1 \uplus D_2$ of dependency pairs is non-looping if D_2 is non-looping, and there is a reduction triple $(>, \geq, \geq_1)$ such that*

- *$l > p$ for all $l \rightsquigarrow p \in D_1$,*
- *$l \geq p$ for all $l \rightsquigarrow p \in D_2$,*
- *$l \geq_1 r$ for all $l \to r \in \mathcal{R}$,*
- *either D is non-collapsing or $\geq$ satisfies the limited subterm property.*

Proof Sketch. If D is looping it has an infinite chain which (as D_2 is non-looping) contains infinitely many pairs in D_1. If D is non-collapsing we can find such a chain without `beta` steps, and have $s_i \geq t_i \geq s_{i+1}$ for all i, and if $\rho_i \in D_1$ even $s_i > t_i$, contradicting well-foundedness of $>$. If D is collapsing then let $[(\rho_i, s_i, t_i)|i \in \mathbb{N}|i \geq j]$ be an infinite dependency chain over D; if $\rho_j \in D_1$ then $s_j > t_j \geq s_{j+1}$, if $\rho_j \in D_2$ then $s_j \geq t_j \geq s_{j+1}$ and if $\rho_j =$ `beta` then there is some substitution δ such that $s_j \geq t_j\delta \geq s_{j+1}\delta$. Since $[(\rho_i, s_i\delta, t_i\delta)|i \in \mathbb{N}|i \geq j+1]$ is also a dependency chain we can continue this reasoning recursively, obtaining a decreasing $\geq$ sequence with infinitely many $>$ steps, contradicting well-foundedness. ◀

Theorem 6.2 can be used to prove that every cycle in the dependency graph approximation of an AFS is non-looping; termination follows with Lemma 5.2. See also Section 9 for an algorithm. For left-linear AFSs, we even have a characterization of termination.

▶ **Theorem 6.3.** *A left-linear AFS with dependency graph approximation G is terminating if and only if for every cycle in G the requirements of Theorem 6.2 are satisfied.*

▶ **Example 6.4.** Termination of twice is proved if there is a reduction triple $(>, \geq, \geq_1)$ with the limited subterm property, such that $l \geq_1 r$ for all rules, and $l > p$ for every dependency pair in $\mathcal{C}_{\mathsf{twice}}$ from Example 5.3 (choosing $D_2 = \emptyset$, which is non-looping).

6.1 Type Changing

The situation so far is not completely satisfactory, because both $>$ and $\geq$ may have to compare terms of different types. Consider for example the dependency pair $\mathsf{twice}^\#(F) \rightsquigarrow F \cdot c_y$ from twice where the two sides have a different type. Moreover, the comparison in the definition of limited subterm property may concern terms of different types. This is problematic because term orderings do not usually compare terms of arbitrary different types; neither any version of the higher order path ordering [11, 5] nor monotone algebras [21] are equipped for this.

A solution is to manipulate the ordering requirements. Let $(\succ, \succeq)$ be a reduction pair (so a pair such that $(\succ, \succeq, \succeq)$ is a reduction triple). Define $>$, $\geq$, and $\geq_1$ as follows:

- $s > t$ if there are fresh variables $x_1, \dots, x_n$ and terms $u_1, \dots, u_m$ such that $s \cdot x_1 \cdots x_n \succ t \cdot u_1 \cdots u_m$ and both sides have some base type;
- $s \geq t$ if there are fresh variables $x_1, \dots, x_n$ and terms $u_1, \dots, u_m$ such that $s \cdot x_1 \cdots x_n \ R \ t \cdot u_1 \cdots u_m$ and both sides have some base type, where R is $\succeq \cup \succ \cdot \succeq \cup \succeq \cdot \succ$;
- $s \geq_1 t$ if $s \succeq t$ and s, t have the same type.

▶ **Lemma 6.5.** *$(>, \geq, \geq_1)$ as generated from a reduction pair $(\succ, \succeq)$ is a reduction triple.*

Proof. This is easy, noting: (1) if $s \geq_1 t$ then by monotonicity $s\vec{x} \succeq t\vec{x}$, (2) if $s > t$ then for any $\vec{u}$ there are $\vec{v}$ such that $s \cdot \vec{u} \succ t \cdot \vec{v}$ (by stability of $\succ$), (3) similar for $\geq$. ◀

The relations $>$ and $\geq$ are not necessarily computable, but we will not need to work with them directly. To prove some set of dependency pairs D non-looping, we can choose for every pair $l \rightsquigarrow p \in D$ a corresponding base-type pair $\overline{l} \rightsquigarrow \overline{p}$, and prove either $\overline{l} \succ \overline{p}$ or $\overline{l} \succeq \overline{p}$. For example, we could assign $\overline{l} := l \cdot x_1 \cdots x_n$ and $\overline{p} := p \cdot c_{y_1} \cdots c_{y_m}$, where the c_{y_i} are chosen arbitrarily. This is the choice we will use in examples in this paper. Other choices for $\overline{p}$, for instance made in such a way as to duplicate existing requirements, are also possible.

To make sure that $\geq$ satisfies the limited subterm property, we consider a base-type version of subterm reduction, which is strongly related to β-reduction.

▶ **Definition 6.6.** $\rhd^!$ is the relation on base-type terms (and $\unrhd^!$ its reflexive closure) generated by the following clauses:

- $(\lambda x.\, s) \cdot t_0 \cdots t_n \rhd^! u$ if $s[x := t_0] \cdot t_1 \cdots t_n \unrhd^! u$
- $f(s_1, \dots, s_m) \cdot t_1 \cdots t_n \unrhd^! u$ if $s_i \cdot \vec{c} \unrhd^! u$
- $s \cdot t_1 \cdots t_n \unrhd^! u$ if $t_i \cdot \vec{c} \unrhd^! u$ (s may have any form)

Here, $s \cdot \vec{c}$ is a term s applied to constants c_y of the right type. Note that if $s \rhd t$ and s has base type, there are terms $u_1, \dots, u_n$ and substitution γ such that $s \rhd^! t\gamma \cdot u_1 \cdots u_n$. Consequently, $\geq$ satisfies the limited subterm property if $\succ \cup \succeq$ contains $\rhd^!$ and $f(\vec{x}) \succeq f^\#(\vec{x})$ for all $f \in \mathcal{D}$ (the *marking property*). We can derive the following theorem.

▶ **Theorem 6.7.** *A set of dependency pairs $D = D_1 \uplus D_2$ is non-looping if D_2 is non-looping and there is a reduction pair $(\succ, \succeq)$ such that:*

1. *$\overline{l} \succ \overline{p}$ for all $l \rightsquigarrow p \in D_1$;*
2. *$\overline{l} \succeq \overline{p}$ for all $l \rightsquigarrow p \in D_2$;*
3. *$l \succeq r$ for all $l \to r \in \mathcal{R}$;*
4. *if D is collapsing, then $\succ \cup \succeq$ contains $\rhd^!$, and $f(\vec{x}) \succeq f^\#(\vec{x})$ for all $f \in \mathcal{D}$.*

▶ **Example 6.8.** To prove that $\mathcal{C}_{\mathsf{twice}}$ is non-looping it suffices to find a reduction pair $(\succ, \succeq)$ such that $l \succeq r$ for all rules, $\succeq$ satisfies the subterm and marking properties, and furthermore:

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{I}^\#(\mathsf{s}(n)) & \succ & \mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot n & \mathsf{twice}^\#(F) \cdot x & \succ & F \cdot (F \cdot c_y) \\
\mathsf{I}^\#(\mathsf{s}(n)) & \succ & \mathsf{twice}^\#(\lambda x.\, \mathsf{I}(x)) \cdot c_z & \mathsf{twice}^\#(F) \cdot x & \succ & F \cdot c_y \\
\mathsf{twice}(F) \cdot m & \succ & F \cdot (F \cdot m) & \mathsf{twice}(F) \cdot m & \succ & F \cdot m
\end{array}$$

This completes the basis of dynamic dependency pairs for AFSs. But is this approach any easier than proving $l > r$ for all rewrite rules? Unless the dependency graph has no cycles we still have to prove $l \geq r$ for all rules and with an ordering like HORPO [11] this is barely an improvement. In Section 7 we will therefore discuss a variation of usable rules, which allows us to drop a number of ordering requirements. In Section 8 we will define a variation of the monotone algebra approach that is especially suited to dependency pairs.

7 Formative Rules

In the first order setting, the result corresponding with Theorem 6.2 is optimized: it is sufficient to consider for a cycle only its *usable rules* instead of all rules. The definition of usable rules cannot easily be extended to our setting, because we admit collapsing dependency pairs. Therefore we take a different approach with the same goal of restricting attention to rules which are in some way relevant to a set of dependency pairs. Where usable rules are defined from the right-hand sides of dependency pairs, our *formative rules* are based on the left-hand sides. We will use the notion of *simple terms*:

▶ **Definition 7.1.** A term s is *simple* if:

- it is linear,
- it has no subterm of the form $x \cdot s_1 \cdots s_n$ with $n > 0$ and x a free variable,
- there is no occurrence of a free variable below an abstraction.

Many examples of AFSs, such as rules from functional programming, have a simple left-hand side. The intuition behind formative rules is that, for rewrite rules with a simple left-hand side, only the formative rules can contribute to the creation of its pattern.

▶ **Definition 7.2.** For β-normal terms s, let $Symb(s)$ be recursively defined as follows:

$$\begin{aligned}
Symb(\lambda x.\, s : \sigma) &= \{\langle ABS, \sigma\rangle\} \cup Symb(s) \\
Symb(f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m : \sigma) &= \{\langle f, \sigma\rangle\} \cup Symb(s_1) \cup \ldots \cup Symb(s_m) \\
Symb(x \cdot s_1 \cdots s_n : \sigma) &= \{\langle VAR, \sigma\rangle\} \cup Symb(s_1) \cup \ldots \cup Symb(s_n) \ (n > 0) \\
Symb(x) &= \emptyset
\end{aligned}$$

The formative symbols and rules of any term are defined by a (possibly) infinite process:

- the starting point: $FS_0(s) = Symb(s)$
- for all $n \geq 0$, the set $FR_n(s)$ consists of rules $l \cdot x_1 \cdots x_k \to r \cdot x_1 \cdots x_k$ if $l \to r \in \mathcal{R}$ and
  - $k = 0,\ r = \lambda x.\, r' : \sigma$ and $\langle ABS, \sigma\rangle \in FS_n(s)$, or
  - $r = f(\vec{u}) \cdot \vec{v} : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_k \Rightarrow \tau$ and $\langle f, \tau\rangle \in FS_n(s)$, or
  - $r = x \cdot \vec{v} : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_k \Rightarrow \tau$ and $\langle f, \tau\rangle \in FS_n(s)$ for some $f \in \mathcal{F} \cup \{ABS, VAR\}$; $|\vec{v}| \geq 0$
- $FS_{n+1}(s) = FS_n(s) \cup \bigcup_{l \to r \in FR_n(s)} Symb(l)$

Now $FR(s)$ is defined as the union of all $FR_n(s)$ (this is a finite union for finite AFSs) in the case that both s is simple and all rules in this union have a simple left-hand side. Otherwise, $FR(s) = \mathcal{R}$. The set of formative rules of a dependency pair, $FR(f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m \rightsquigarrow p)$, is defined as $\bigcup_{1 \leq i \leq m} FR(l_i)$. For a set D of dependency pairs, $FR(D) = \bigcup_{l \rightsquigarrow p \in D} FR(l \rightsquigarrow p)$.

Note that $FR_n(s)$ and $FS_{n+1}(s)$ can easily be calculated (automatically) from $FS_n(s)$; to compute $FR(s)$ a tool would simply repeat this process until either a rule with a non-simple left-hand-side is included (in which case $FR(s) = \mathcal{R}$), or until no new symbols are added.

▶ **Example 7.3.** Recall the rules for the (completed) system twice:

$$\begin{array}{llllllll} (A) & \mathsf{I}(\mathsf{o}) & \to & \mathsf{o} & (C) & \mathsf{twice}(F) & \to & \lambda y. F \cdot (F \cdot y) \\ (B) & \mathsf{I}(\mathsf{s}(n)) & \to & \mathsf{s}(\mathsf{twice}(\lambda x. \mathsf{I}(x)) \cdot n) & (D) & \mathsf{twice}(F) \cdot m & \to & F \cdot (F \cdot m) \end{array}$$

In this context, let $l = \mathsf{s}(n)$. Then

$$\begin{array}{lllllll} FS_0(l) & = & \{\langle \mathsf{s}, \mathsf{nat}\rangle\} & & FS_1(l) & = & \{\langle \mathsf{s}, \mathsf{nat}\rangle, \langle \mathsf{twice}, \mathsf{nat}\rangle, \langle \mathsf{I}, \mathsf{nat}\rangle\} \\ FR_0(l) & = & \{(B),\ (D)\} & & FR_1(l) & = & \{(B),\ (D)\} = FR_0(l) \end{array}$$

We have $FR(\mathsf{I}^\#(\mathsf{s}(n)) \rightsquigarrow p) = FR(\mathsf{s}(n)) = \{(B),\ (D)\}$ for any p. Note that for a dependency pair with left-hand side $\mathsf{twice}(F) \cdot n$ or $\mathsf{twice}^\#(F)$ the set of formative rules is empty (since $Symb(F) = Symb(n) = \emptyset$). Therefore, the formative rules of the SCC $\mathcal{C}_{\mathsf{twice}}$ are (B) and (D).

Using formative rules Formative rules are constructed in such a way that to reduce to a term of the form $l\gamma$ we only need its formative rules:

▶ **Lemma 7.4.** *If s is terminating and $s \to_{\mathcal{R}}^* l\gamma$, then there exists a substitution δ on the same domain as γ such that each $\delta(x) \to_{\mathcal{R}}^* \gamma(x)$ and $s \to_{FR(l)}^* l\delta$.*

Proof Sketch. We assume l is simple and not a variable (otherwise this is trivial). Transform the reduction $s \to_{\mathcal{R}}^* l\gamma$ into a reduction without any headmost steps with a rule $l' \to \lambda x. r'$ (this is possible because the rules have been completed). Then perform induction on s first, using $\to_{\mathcal{R}} \cup \rhd$, the length of the reduction second. If s is headed by a beta-redex we can start with a β-step because $l\gamma$ is not (and complete with IH1), if s reduces to $l\gamma$ without any headmost steps we use the $\rhd$ part of IH1 (variable capture is not an issue because γ can be assumed to have empty domain if l is an abstraction) and if $s \to_{\mathcal{R}}^* l'\gamma' \cdot t_1 \cdots t_n \to_{\mathcal{R}} r'\gamma' \cdot t_1 \cdots t_n \to_{\mathcal{R}'}^* l\gamma$ with either r' headed by a variable or the latter part not using any headmost steps, then $l'' := l' \cdot x_1 \cdots x_n \to r' \cdot x_1 \cdots x_n =: r''$ is a formative rule of l and can be assumed simple, so we use the second induction hypothesis to get $s \to_{FR(l'')}^* l''\delta' \to_{\mathcal{R}} r''\delta'$ and the first induction hypothesis to have $r''\delta' \to_{FR(l)}^* l\delta$; this suffices because $FR(l'') \subseteq FR(l)$. ◀

With this we can strengthen the definition of dependency chains, and adapt Theorem 4.5:

▶ **Lemma 7.5.** *If $\mathcal{R}$ is non-terminating, there is an infinite dependency chain over* $\mathsf{DP}(\mathcal{R})$ *such that for all i: $t_i \to_{in}^* s_{i+1}$ using only rules from $FR(l_{i+1})$.*

Thus, we can restrict attention to dependency chains using only formative rules, and adapt the definition of *looping* and the results of Sections 5 and 6 accordingly. We obtain:

▶ **Theorem 7.6** (Complete Result). *A set of dependency pairs $D = D_1 \uplus D_2$ is non-looping if D_2 is non-looping and there is a reduction triple such that:*

1. *$l > p$ for $l \rightsquigarrow p \in D_1$,*
2. *$l \geq p$ for $l \rightsquigarrow p \in D_2$,*
3. *$l \geq_1 r$ for $l \to r \in FR(D)$,*
4. *If D is collapsing, then $\geq$ additionally satisfies the limited subterm property.*

Also $\emptyset$ is non-looping. An AFS with rules $\mathcal{R}$ and dependency graph approximation G is terminating if all cycles in G are non-looping, which holds if all SCCs are non-looping.

In requirement (3) in Theorem 6.7 we can also restrict attention to the formative rules of D instead of considering all rules. It remains to find a suitable reduction triple or pair.

8 Monotone Algebras

A semantical method to prove termination of rewriting is to interpret terms in a well-founded algebra, and show that whenever $s \to t$ their interpretations decrease: $[\![s]\!] > [\![t]\!]$. For TRSs, such an algebra is called a termination model if $[\![l]\!] > [\![r]\!]$ for all rules $l \to r$ and some additional properties guarantee that this implies $[\![C[l\gamma]]\!] > [\![C[r\gamma]]\!]$ for all contexts C and substitutions γ. A TRS is terminating if and only if it has a termination model [9, 26]. Van de Pol [21] generalizes this approach to HRSs, with higher order rewriting modulo $\alpha\beta\eta$, and shows that a HRS is terminating if it has a termination model; the converse does not hold.

Here we consider interpretations of AFS terms in a monotone algebra, and use the orderings to solve dependency pair constraints. Since $>$ does not have to be monotonic when using dependency pairs, the theory of [21] can be significantly simplified. We interpret all base types with the same algebra to avoid problems with comparing differently-typed terms.

▶ **Definition 8.1** (Weakly Monotonic Functionals)**.** Let $\mathcal{A}$ be an algebra with a well-founded partial order $>$ and minimum element 0. We assume there is a binary operator $\vee$ on $\mathcal{A}$ such that $x \vee y \geq x, y$ for all $x, y \in \mathcal{A}$ and $x \vee 0 = x$. Terms will be interpreted by elements of, and weakly monotonic functionals over, $\mathcal{A}$. Intuitively, a functional f is weakly monotonic if $f(x) \geq f(y)$ whenever $x \geq y$; however, f only needs to be defined on weakly monotonic input. We inductively define the weakly monotonic functionals for all types, and relations $\sqsupset_{wm}$ and $\sqsupseteq_{wm}$ on these functionals:

- the interpretation for base types: $\mathcal{WM}_\iota = \mathcal{A}$ for all $\iota \in \mathcal{B}$,
- the orderings on $\mathcal{WM}_\iota$ (with $\iota \in \mathcal{B}$): $\sqsupset_{wm}$ equals $>$, and $\sqsupseteq_{wm}$ is its reflexive closure,
- the interpretation for functional types: $\mathcal{WM}_{\sigma \Rightarrow \tau}$ consists of the functions mapping elements of $\mathcal{WM}_\sigma$ to elements of $\mathcal{WM}_\tau$, such that $\sqsupseteq_{wm}$ is preserved (that is, if $x \sqsupseteq_{wm} y$ in $\mathcal{WM}_\sigma$ then $f(x) \sqsupseteq_{wm} f(y)$ in $\mathcal{WM}_\tau$),
- the orderings on $\mathcal{WM}_{\sigma \Rightarrow \tau}$: we have $f \sqsupset_{wm} g$ iff $f(x) \sqsupset_{wm} g(x)$ for all $x \in \mathcal{WM}_\sigma$, and $f \sqsupseteq_{wm} g$ iff $f(x) \sqsupseteq_{wm} g(x)$ for all $x \in \mathcal{WM}_\sigma$.

$\sqsupset_{wm}$ and $\sqsupseteq_{wm}$ are an order and quasi-order respectively, and strongly compatible. If either $x \sqsupset_{wm} y$ or $x = y$ then $x \sqsupseteq_{wm} y$, but the converse implication does not hold.

Constant functions are weakly monotonic functionals: for $n \in \mathcal{A}$ and $\sigma = \tau_1 \Rightarrow \ldots \Rightarrow \tau_k \Rightarrow \iota$ (note that ι always refers to a base type), let $n_\sigma = \boldsymbol{\lambda} x_1 \ldots x_k.n$ (the function in $\mathcal{WM}_\sigma$ taking k arguments and returning n). The function $\boldsymbol{\lambda} f.f(\vec{0})$ is also in $\mathcal{WM}_{\sigma \Rightarrow \iota}$, where $f(\vec{0})$ is short for $f(0_{\tau_1}, \ldots, 0_{\tau_k})$. A weakly monotonic functional not defined in [21], but which will be needed to deal with term application, is *max*:

$$\begin{array}{rcll} \max_\iota(x, y) & = & x \vee y & (\text{for } x, y \in \mathcal{A}) \\ \max_{\sigma \Rightarrow \tau}(f, y) & = & \boldsymbol{\lambda} x.\max_\tau(f(x), y) & (\text{for } f \in \mathcal{WM}_{\sigma \Rightarrow \tau}, y \in \mathcal{A}) \end{array}$$

Using induction on the type of the first argument, it is easy to see that $\max_\sigma \in \mathcal{WM}_{\sigma \Rightarrow \iota \Rightarrow \sigma}$.

Term Interpretation. Using an interpretation $\mathcal{J}$ of function symbols, van de Pol associates to each closed term a weakly monotonic functional. Although the definition in [21] considers terms modulo $\alpha\beta\eta$, this is not a significant blockade because we can handle application as a function symbol. The following is our own adaptation of the translation in [21]:

▶ **Definition 8.2.** For all function symbols $f : (\sigma_1 \times \ldots \times \sigma_n) \Rightarrow \tau$ let $\mathcal{J}_f \in \mathcal{WM}_\sigma$, where σ is $\sigma_1 \Rightarrow \ldots \sigma_n \Rightarrow \tau$. A valuation is a function α with a finite domain of variables, such that $\alpha(x) \in \mathcal{WM}_\sigma$ for $x : \sigma$ in its domain. For any AFS-term s and valuation α whose domain contains all $x \in FV(s)$, let $[\![s]\!]_{\mathcal{J},\alpha}$ be the weakly monotonic functional defined as follows:

$$\begin{array}{lcl}
[\![x]\!]_{\mathcal{J},\alpha} & = & \alpha(x) \ \text{ if } x \in \mathcal{V} \\
[\![f(s_1,\dots,s_n)]\!]_{\mathcal{J},\alpha} & = & \mathcal{J}_f([\![s_1]\!],\dots,[\![s_n]\!]) \\
[\![\lambda x.\, s]\!]_{\mathcal{J},\alpha} & = & \boldsymbol{\lambda} n.[\![s]\!]_{\mathcal{J},\alpha\cup\{x\mapsto n\}} \ \text{ if } x \notin \mathsf{dom}(\alpha) \\
[\![s \cdot t]\!]_{\mathcal{J},\alpha} & = & \max([\![s]\!]_{\mathcal{J},\alpha}([\![t]\!]_{\mathcal{J},\alpha}), [\![t]\!]_{\mathcal{J},\alpha}(\vec{0}))
\end{array}$$

▶ **Example 8.3.** In our running example, consider an interpretation into the natural numbers Say $\mathcal{J}_{\mathsf{l}} = \boldsymbol{\lambda} n.n$ and $\mathcal{J}_{\mathsf{s}} = \boldsymbol{\lambda} n.n+1$. Then $[\![\mathsf{l}(\mathsf{s}(x))]\!]_{\mathcal{J},\alpha} = \alpha(x)+1$.

Reduction Pair Since this definition uses weak rather than strict monotonicity it cannot be used directly like in first order rewriting: $[\![l]\!] \sqsupset_{wm} [\![r]\!]$ does not in general imply $[\![C[l\gamma]]\!] \sqsupset_{wm} [\![C[r\gamma]]\!]$. This issue (which van de Pol works around by defininig an additional relation) disappears in the context of dependency pairs. Using Theorem 6.7 we obtain a number of requirements $[\![l]\!] \sqsupset_{wm} [\![r]\!]$ or $[\![l]\!] \sqsupseteq_{wm} [\![r]\!]$, and additionally, for collapsing D, the subterm and marking properties must be satisfied. The latter is a simple restriction, the former holds if the value of a function is always greater than or equal to the value of its arguments.

▶ **Theorem 8.4.** *Let $\mathcal{J}$ be a symbol interpretation such that:*

- *$\mathcal{J}_f \sqsupseteq_{wm} \mathcal{J}_{f^\#}$ for all $f \in \mathcal{D}$*
- *$\mathcal{J}$ maps each c_x to the appropriate 0_σ*
- *for all $f : (\sigma_1 \times \dots \times \sigma_n) \Rightarrow \tau_1 \Rightarrow \dots \Rightarrow \tau_m \Rightarrow \iota \in \mathcal{F}$, all $1 \le i \le n$ and all $n \in \mathcal{WM}_{\sigma_i}$: $\mathcal{J}_f(0_{\sigma_1},\dots,n,\dots,0_{\sigma_n},0_{\tau_1},\dots,0_{\tau_m}) \sqsupseteq_{wm} n(\vec{0})$.*

Define $s \succ t$ if $[\![s]\!]_{\mathcal{J},\alpha} \sqsupset_{wm} [\![t]\!]_{\mathcal{J},\alpha}$ for all valuations α and $s \succeq t$ if $[\![s]\!]_{\mathcal{J},\alpha} \sqsupseteq_{wm} [\![t]\!]_{\mathcal{J},\alpha}$ for all valuations α. Then $(\succ,\succeq)$ is a reduction pair which satisfies the subterm and marking properties from Theorem 6.7.

Proof. Compatibility is evident, weak monotonicity holds by a simple case distinction and stability by the substitution Lemma [21, Theorem 3.2.1]. By the interpretation of application also $\rightarrow_\beta$ is contained in $\succeq$, and subterm reduction is included by an inductive argument which uses the last two requirements. The marking property is given by the first requirement. ◀

It is not immediately obvious how to use monotone algebras automatically; a lot will depend on the chosen interpretation for the function symbols. Common first order methods, like polynomial or matrix interpretations, are not likely to be succesful in the presence of functional variables. However, it is very likely that higher order parallels exist, such as an interpretation with primitive recursive functions. While a proper study of such methods is beyond the scope of this paper, the example below might give some initial ideas.

▶ **Example 8.5.** Suppose we have to satisfy a requirement $\mathsf{map}(F,\mathsf{cons}(x,y)) \succeq \mathsf{cons}(F \cdot x, \mathsf{map}(F,y))$, where $F : \mathsf{nat} \Rightarrow \mathsf{nat}$. We consider an interpretation in the natural numbers (with standard $>$, 0, and $\vee$ giving the highest of two numbers) using primitive recursive functions. Let $G(f,m,n)$ be the recursive function defined by: $G(f,m,0) = \max(f(m),m)$ and $G(f,m,n{+}1) = f(n{+}1, 2G(f,m,n))$. This function is weakly monotonic in each of f, m and n, and moreover $G(f,m,n{+}k) \ge G(f,m,n){+}G(f,m,k)$ for all $n,k > 0$. Also $G(f,m,n) \ge m$, and $G(f,m,n) \ge f(0)$ if f is weakly monotonic. Choose $\mathcal{J}_{\mathsf{cons}} = \boldsymbol{\lambda} nm.n{+}m{+}1$ and $\mathcal{J}_{\mathsf{map}} = \boldsymbol{\lambda} fn.G(f,n,n{+}1)$, and let $\alpha = \{F \mapsto f, x \mapsto n, y \mapsto m\}$ be a valuation. Then:

$$\begin{array}{lcl}
[\![\mathsf{map}(F,\mathsf{cons}(x,y))]\!]_{\mathcal{J},\alpha} & = & G(f,n{+}m{+}1,n{+}m{+}2) \\
& \sqsupseteq_{wm} & G(f,n{+}m{+}1,n{+}1) + G(f,n{+}m{+}1,m{+}1) \\
& = & f(n{+}1) + 2G(f,n{+}m{+}1,n) + G(f,n{+}m{+}1,m{+}1) \\
& \sqsupseteq_{wm} & f(n) + 2(n{+}m{+}1) + G(f,m,m{+}1) \\
& \sqsupset_{wm} & \max(f(n),n) + G(f,m,m{+}1) + 1 \\
& = & [\![\mathsf{cons}(F \cdot x, \mathsf{map}(F,y))]\!]_{\mathcal{J},\alpha}
\end{array}$$

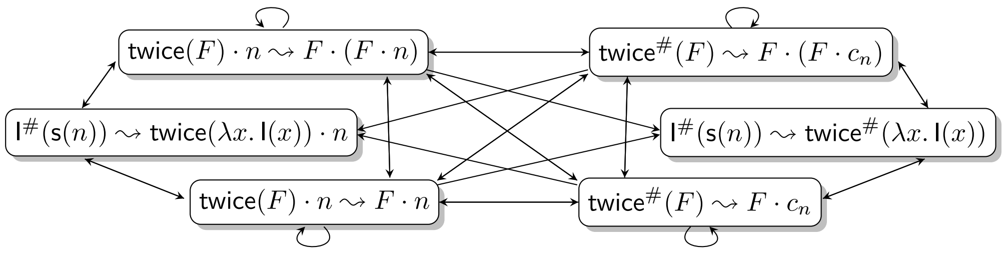
9 Conclusion

A Termination Algorithm The combination of Theorem 7.6 and Section 6.1 provides an algorithm to prove termination of an AFS. First calculate the system's dependency pairs and take an approximation of the (finite) dependency graph. Then:

1. remove all nodes from G which are not on a cycle;
2. if G is empty return `terminating`; otherwise find an SCC $\mathcal{C}$;
3. determine a partition in $\mathcal{C} = \mathcal{C}_1 \uplus \mathcal{C}_2$ and find a reduction pair $(\succ, \succeq)$ such that $\overline{l} \succ \overline{p}$ for $l \rightsquigarrow p \in \mathcal{C}_1$, $\overline{l} \succeq \overline{p}$ for $l \rightsquigarrow p \in \mathcal{C}_2$, $l \succeq r$ for $l \to r \in FR(\mathcal{C})$ and either $\mathcal{C}$ is non-collapsing, or $\succ \cup \succeq$ contains $\rhd^!$ and $f(\vec{x}) \succeq f^\#(\vec{x})$ for all $f \in \mathcal{D}$; if this step fails, return `fail`;
4. remove all pairs in $\mathcal{C}_1$ from the graph, since any cycle $\mathcal{C}'$ which includes such a pair is a subcycle of $\mathcal{C}$ and thus also proved non-looping by $(\succ, \succeq)$; continue with (1).

The algorithm iterates over a graph approximation, simplifying SCCs until none remain; note that this moves in the direction of the dependency pair framework as defined in [6].

▶ **Example 9.1.** Consider our running example twice, whose dependency graph was shown in Example 5.1. As instructed in step (1) of the algorithm, we remove nodes not on a cycle.



$\mathsf{twice}(F) \cdot n \rightsquigarrow F \cdot (F \cdot n)$
$\mathsf{twice}^\#(F) \rightsquigarrow F \cdot (F \cdot c_n)$
$\mathsf{I}^\#(\mathsf{s}(n)) \rightsquigarrow \mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot n$
$\mathsf{I}^\#(\mathsf{s}(n)) \rightsquigarrow \mathsf{twice}^\#(\lambda x.\, \mathsf{I}(x))$
$\mathsf{twice}(F) \cdot n \rightsquigarrow F \cdot n$
$\mathsf{twice}^\#(F) \rightsquigarrow F \cdot c_n$

In step (2) we choose the SCC of all pairs in the graph; its formative rules are calculated in Example 7.3. For step (3) let $\mathcal{C}_1 := \{\mathsf{I}^\#(\mathsf{s}(n)) \rightsquigarrow \mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot n, \mathsf{I}^\#(\mathsf{s}(n)) \rightsquigarrow \mathsf{twice}^\#(\lambda x.\, \mathsf{I}(x))\}$ and $\mathcal{C}_2$ the set containing the other pairs. We have the following proof obligations:

$$\begin{array}{llcllllcl}
A. & \mathsf{I}^\#(\mathsf{s}(n)) & \succ & \mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot n & & E. & \mathsf{twice}^\#(F) \cdot x & \succeq & F \cdot (F \cdot c_y) \\
B. & \mathsf{I}^\#(\mathsf{s}(n)) & \succ & \mathsf{twice}^\#(\lambda x.\, \mathsf{I}(x)) \cdot c_z & & F. & \mathsf{twice}^\#(F) \cdot x & \succeq & F \cdot c_y \\
C. & \mathsf{twice}(F) \cdot m & \succeq & F \cdot (F \cdot m) & & G. & \mathsf{I}(\mathsf{s}(n)) & \succeq & \mathsf{s}(\mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot n) \\
D. & \mathsf{twice}(F) \cdot m & \succeq & F \cdot m & & H. & \mathsf{twice}(F) \cdot m & \succeq & F \cdot (F \cdot m)
\end{array}$$

Requirement (H) is a duplicate of (C). Using an interpretation in functionals over the natural numbers where each $\mathcal{J}_{c_x} = 0$, and assuming $\mathcal{J}_{\mathsf{twice}} = \mathcal{J}_{\mathsf{twice}^\#}$, (B) is implied by (A), and (E) by (C), and (F) by (D). The remaining requirements are satisfied with $\mathcal{J}_{\mathsf{I}^\#} = \mathcal{J}_{\mathsf{I}} = \boldsymbol{\lambda} n.n$ and $\mathcal{J}_{\mathsf{s}} = \boldsymbol{\lambda} n.n + 1$ and $\mathcal{J}_{\mathsf{twice}^\#} = \mathcal{J}_{\mathsf{twice}} = \boldsymbol{\lambda} f.\boldsymbol{\lambda} n.f(f(n))$:

$$\begin{array}{lrcl}
A. & n+1 & > & \max((\boldsymbol{\lambda} n.n)((\boldsymbol{\lambda} n.n)n), n) = \max(n,n) = n \\
C. & \max(F(F(n)), n) & \geq & \max(F(\max(F(n),n)), \max(F(n),n)) \\
D. & \max(F(F(n)), n) & \geq & \max(F(n), n) \\
G. & n+1 & \geq & \max(n,n) + 1 = n+1
\end{array}$$

The calculations for (A) and (G) are obvious. With some reasoning (distinguishing the cases $n > F(n)$, and $F(n) \geq n$ and noting that $F(n) \geq n$ implies $F(F(n)) \geq F(n)$ by weak monotonicity), (C) and (D) also hold.

Thus we move on to step (4) and remove the two nodes in $\mathcal{C}_1$ from the graph:

$$\begin{tikzcd}
\boxed{\mathsf{twice}(F)\cdot n \rightsquigarrow F\cdot(F\cdot n)} \arrow[loop left] \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow] & \boxed{\mathsf{twice}^{\#}(F) \rightsquigarrow F\cdot(F\cdot c_n)} \arrow[loop right] \arrow[d, leftrightarrow] \arrow[dl, leftrightarrow] \\
\boxed{\mathsf{twice}(F)\cdot n \rightsquigarrow F\cdot n} \arrow[loop left] \arrow[r, leftrightarrow] & \boxed{\mathsf{twice}^{\#}(F) \rightsquigarrow F\cdot c_n} \arrow[loop right]
\end{tikzcd}$$

All nodes are still interconnected, so we continue with the SCC of all pairs. Interestingly, $FR(\mathcal{C}) = \emptyset$. Therefore it suffices to find a reduction pair with the usual properties and:

$$\begin{array}{lcllcl}
\mathsf{twice}(F)\cdot n & \succ & F\cdot(F\cdot n) & \mathsf{twice}^{\#}(F)\cdot n & \succ & F\cdot(F\cdot c_y) \\
\mathsf{twice}(F)\cdot n & \succ & F\cdot n & \mathsf{twice}^{\#}(F)\cdot n & \succ & F\cdot c_y
\end{array}$$

This is satisfied with an algebra interpretation with $\mathcal{J}_{\mathsf{twice}^{\#}} = \mathcal{J}_{\mathsf{twice}} = \boldsymbol{\lambda} fn.\max(f(f(n)), n) + 1$. Thus we remove the final four nodes from the graph, and conclude that twice is terminating.

Summary and Future Work We have defined a first basic dependency pair method for AFSs, with a variation of usable rules which takes into account the possible presence of collapsing dependency pairs. We have explained that besides orderings such as HORPO also monotone algebras can be used to solve the ordering constraints.

We intend to further study dependency pairs for AFSs with restrictions. For example, if function symbols have a base output type we can drop requirements, yielding an easier method. If we restrict to rules without abstractions in the left-hand sides, we may weaken the subterm property to obtain a stronger method, and define for instance argument filterings (in the extended abstract [15] a first step in this direction is given for HRSs).

A preliminary version of the dependency pair method with argument filterings is implemented in the tool WANDA v1.0 [14]. We intend to improve the implementation by taking into account also the dependency graph, strongly connected components and formative rules.

This work aims to contribute to the larger goal of understanding dependency pairs for higher order rewriting, and creating tools to automatically prove termination in this setting.

Acknowledgments. We are very grateful for the constructive comments of the referees that helped to improve the paper. We also gratefully acknowledge remarks from Jan Willem Klop.

References

1 T. Aoto and Y. Yamada. Dependency pairs for simply typed term rewriting. In J. Giesl, editor, *Proceedings of RTA 2005*, volume 3467 of *LNCS*, pages 120–134, Nara, Japan, April 2005. Springer.

2 T. Aoto and Y. Yamada. Argument filterings and usable rules for simply typed dependency pairs. In S. Ghilardi and R. Sebastiani, editors, *Proceedings of FroCoS 2009*, volume 5749 of *LNAI*, pages 117–132, Trento, Italy, September 2009. Springer.

3 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

4 F. Blanqui. Higher-order dependency pairs. In *Proceedings of WST 2006*, Seattle, USA, August 2006.

5 F. Blanqui, J.-P. Jouannaud, and A. Rubio. The computability path ordering: The end of a quest. In *CSL 2008*, volume 5213 of *LNCS*, pages 1–14, Bertinoro, Italy, July 2008. Springer.

6 J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proceedings of LPAR 2004*, volume 3452 of *Lecture Notes in Computer Science*, pages 301–331. Springer, 2005.

7 N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 205(4):474–511, 2007.

8 Nao Hirokawa, Aart Middeldorp, and Harald Zankl. Uncurrying for termination. In *LPAR 2008*, volume 5330 of *LNAI*, pages 667–681, Doha, 2008. Springer-Verlag.

9 G. Huet and D.C. Oppen. Equations and rewrite rules: a survey. In R.V Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, London, 1980.

10 J.-P. Jouannaud and M. Okada. A computation model for executable higher-order algebraic specification languages. In *LICS 1991*, pages 350–361, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

11 J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *LICS 1999*, pages 402–411, Trento, Italy, July 1999.

12 Z. Khasidashvili. Expression Reduction Systems. In *Proceedings of I. Vekua Institute of Applied Mathematics*, volume 36, pages 200–220, Tblisi, Georgia, 1990.

13 J.W. Klop. *Combinatory Reduction Systems*, volume 127 of *Mathematical Centre Tracts*. CWI, Amsterdam, The Netherlands, 1980. PhD Thesis.

14 C. Kop. Wanda. `http://www.few.vu.nl/~kop/code.html`.

15 C. Kop and F. van Raamsdonk. Higher-order dependency pairs with argument filterings. In *Proceedings of WST 2010*, Edinburgh, UK, July 2010. `http://www.few.vu.nl/~kop/wst10.pdf`.

16 K. Kusakari. On proving termination of term rewriting systems with higher-order variables. *IPSJ Transactions on Programming*, 42(SIG 7 PRO11):35–45, 2001.

17 K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE Transactions on Information and Systems*, 92(10):2007–2015, 2009.

18 K. Kusakari and M. Sakai. Enhancing dependency pair method using strong computability in simply-typed term rewriting. *AAECC*, 18(5):407–431, 2007.

19 K. Kusakari and M. Sakai. Static dependency pair method for simply-typed term rewriting and related techniques. *IEICE Transactions*, 2(92-D):235–247, 2009.

20 T. Nipkow. Higher-order critical pairs. In *LICS 1991*, pages 342–349, Amsterdam, The Netherlands, July 1991.

21 J.C. van de Pol. *Termination of Higher-order Rerwite Systems*. PhD thesis, University of Utrecht, 1996.

22 M. Sakai and K. Kusakari. On dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E88-D(3):583–593, 2005.

23 M. Sakai, Y. Watanabe, and T. Sakabe. An extension of the dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E84-D(8):1025–1032, 2001.

24 S. Suzuki, K. Kusakari, and F. Blanqui. Argument filterings and usable rules in higher-order rewrite systems. *IPSJ Transactions on Programming*, 4(2):1–12, 2011. To appear.

25 Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

26 H. Zantema. Termination of term rewriting: interpretation and type elimination. *Journal of Symbolic Computation*, 17:23–50, 1994.