Seminar Report

Higher-Order Dependency Pairs

Julian Nagele
julian.nagele@student.uibk.ac.at

29 February 2012

Supervisor: Univ.-Prof. Dr. Aart Middeldorp

Abstract

A well-known termination technique for first-order rewriting is the dependency pair approach. The presence of higher-order variables and beta-reduction makes lifting the definitions and results to higher-order rewriting a non-trivial task. The inherent differences in the various higher-order formalisms further complicate matters. Consequently many proposals exist, which can roughly be split into the so called static and dynamic dependency pairs. This report summarises the existing literature and provides examples to explain the differences, advantages, disadvantages and peculiarities of the most common approaches.

Contents

1 Introduction

The wish for a higher-order formalism evolves naturally, when considering for example higher-order functions in functional programming languages.

Example 1.1. A standard example of a higher-order function is the map function, here shown in Haskell:

```
map f []     = []
map f (h:t)  = f h : map f t
```

In the second rule f occurs as a variable on the left-hand side, but is also applied to h as a function on the right-hand side. This is not possible in first-order term rewriting as there are no variables for function symbols. The other main motivation for higher-order rewriting is the need to extend term rewriting with bound variables, also a common feature in many programming languages. Moreover the wish for this extension emerges when considering, for instance, predicate logic. Also in mathematics many equations contain bound variables, for example when integrals or derivatives are present. This explains the need for a theory of higher-order rewriting, where the rewrite rules may contain bound variables. Thus higher-order rewriting combines term rewriting and the simply typed lambda calculus.

Higher-order rewriting comes in many different flavours. One of the first are the Combinatory Reduction Systems (CRS) by Klop, 1980 [13]. In later years more formalisms were introduced. Some examples are the Expression Reduction Systems (ERS) of Khasidashvili, 1990 [12], Higher-Order Rewrite Systems (HRS) by Nipkow, 1991 [18] and the similar Higher-Order Term Rewriting Systems by Wolfram, 1991 [23]. Jouannaud and Okada, 1991, defined the Algebraic Functional Systems (AFS) [9].

A powerful, well-known method to show termination of first-order rewrite systems is the dependency pair (DP) approach by Arts and Giesl [1]—many extensions and optimisations have been proposed [6, 7, 8]. Consequently dependency pairs have also been studied as a termination technique for higher-order rewriting. This work can roughly be split into two approaches: the static and the dynamic style. They mainly differ in their treatment of higher-order variables in right-hand sides of rules. In the dynamic approach higher-order variables give rise to a dependency pair, in the static approach they do not. The consequence is that static dependency pairs are closer to the first-order definition, which allows optimisations like argument filterings and usable rules—however they can only be applied to a limited class of systems. Dynamic dependency pairs tend to yield stronger results, but most optimisations are not applicable at all, or require a lot of adaptation.

The remainder of this report is structured as follows. In Section 2 two formalisms for higher-order rewriting—Higher-Order Rewrite Systems and Algebraic Functional Systems—are introduced. Section 3 discusses dependency pairs in a higher-order setting, in particular differences between the static and

the dynamic approach. We then sketch how these dependency pairs can be used to show termination of a rewrite system in Section 4. Finally we conclude in Section 5.

2 Higher-Order Rewriting

Many different formalisms for higher-order rewriting have been proposed, two of which we will consider in this report. Dependency pairs in a higher-order setting are commonly studied using Higher-Order Rewrite Systems. Recently also for Algebraic Functional Systems a dependency pair approach has been proposed—AFSs are currently the only formalism used in the annual termination competition.

We assume familiarity with the basic notions of term rewriting and lambda calculus. Since both HRSs and AFSs can be seen as extensions of the simply typed lambda calculus, we recall the concepts and definitions that we use later on.

2.1 Simply Typed Lambda Calculus ($\lambda^{\to}$)

Simple types are generated from a set of *base types*, also called sorts, and the binary type constructor $\to$ according to the following grammar:

$$\mathbb{T} ::= \mathbb{S} \mid (\mathbb{T} \to \mathbb{T})$$

Following the usual conventions $\to$ is right-associative and outermost parentheses are omitted. We denote types by $\sigma, \tau, \rho, \ldots$ and base types by ι, κ. Throughout the remainder of this report, a set $\mathcal{V}$ of typed variables, containing countably many variables of every type will be assumed. Variables will be denoted by $x : \sigma$, $y : \tau, \ldots$, or just x, y if no confusion can arise. Simply typed lambda terms are built from variables, λ-abstraction and application.

Definition 2.1. A term s is a simply typed lambda term, if $s : \sigma$ for some type σ can be inferred using the following rules:

$$\begin{array}{ll} (\text{var})\ x : \sigma & \text{if } x : \sigma \in \mathcal{V} \\ (\text{app})\ u \cdot t : \sigma & \text{if } u : \tau \to \sigma \text{ and } t : \tau \\ (\text{abs})\ \lambda x : \tau . t : \tau \to \rho & \text{if } x : \tau \in \mathcal{V} \text{ and } t : \rho \end{array}$$

The abstraction $\lambda x.s$ *binds* the variable x in s. A variable that is not bound is *free*. We collect all free variables of s in $\mathsf{FV}(s)$ and all bound variables in $\mathsf{BV}(s)$. As usual, application is considered left associative. We denote the *head* of an application as $\mathsf{head}(\cdot)$, i.e., $\mathsf{head}(s \cdot t) = \mathsf{head}(s)$ and $\mathsf{head}(s) = s$ otherwise.

Term equality is modulo α, that is, renaming of bound variables. Hence we tacitly assume bound variables to be fresh whenever necessary—the variable convention.

A *substitution* γ is a type-preserving mapping from variables to terms such that $x : \sigma \neq \gamma(x : \sigma)$ for finitely many $x : \sigma \in \mathcal{V}$. Substitutions are often written

in the form $[x_1 := t_1, \dots x_n := t_n]$. We assume that substituting does not capture free variables. A *context* is a term C containing exactly one occurrence of a special symbol $\Box_\sigma$. The replacement of $\Box_\sigma$ by a term s of type σ is denoted by $C[s]$. Note that here s may contain variables bound in C.

Lambda terms are rewritten using β-reduction:

$$\text{(beta) } C[(\lambda x : \sigma .s) \cdot t] \to_\beta C[s[x : \sigma := t]]$$

It is well-known that beta-reduction is terminating in $\lambda^\to$. Additionally we will use η-expansion. Since η is not terminating in general, we use a restricted version, which guarantees termination. We have

$$\text{(eta) } C[s] \to_\eta C[\lambda x.s \cdot x]$$

- if x is a fresh variable,
- s is of functional type, and
- no β-redex is created.

Using this special η-expansion every term s has a unique η-long, β-normal form, which we denote by $s\updownarrow^\eta_\beta$.

Based on $\lambda^\to$ we now introduce Algebraic Functional Systems and Higher-Order Rewrite Systems.

2.2 Algebraic Functional Systems

The first variant of higher-order rewriting relevant for this report are Algebraic Functional Systems by Jouannaud and Okada [9]. Terms in AFSs combine simply typed λ-terms with algebraic terms. That is, to typed variables, abstraction and application, typed function symbols are added, which have a fixed arity. Every rewrite system has β-reduction as a "built-in" rule. Hence matching is modulo α, which gives rise to the name plain higher-order rewriting, a synonym sometimes used for AFSs.

Definition 2.2. A *type declaration* is an expression of the form $(\sigma_1 \times \cdots \times \sigma_n) \to \sigma$, written just σ if $n = 0$. Type declarations are not types, but are used for typing terms, headed by function symbols.

A *signature* is a set of function symbols $\mathcal{F}$, each equipped with a unique type declaration.

Note that a function symbol $f : (\sigma_1 \times \cdots \times \sigma_n) \to \sigma$ takes exactly n arguments, i.e., has arity n. Moreover σ is not necessarily a base type. Using these additional definitions, terms are defined as follows.

Definition 2.3. The set of *terms* over $\mathcal{V}$ and $\mathcal{F}$, denoted $\mathcal{T}(\mathcal{F}, \mathcal{V})$, is the smallest set consisting of all s, such that $s : \sigma$ for some type σ can be inferred using the clauses (var), (app), (abs) from Definition 2.1 and additionally

$$\text{(fun) } f(s_1, \dots, s_n) : \sigma \quad \text{if } f : (\sigma_1 \times \cdots \times \sigma_n) \to \sigma \in \mathcal{F} \text{ and } s_1 : \sigma_1, \dots, s_n : \sigma_n$$

Definition 2.4. A *rewrite rule* over $\mathcal{F}$ and $\mathcal{V}$ is a pair of terms $l \to r$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$, such that l and r have the same type and all free variables of r occur in l. An AFS consists of a signature $\mathcal{F}$ and a set $\mathcal{R}$ of rules.

Example 1.1 from the introduction can be modelled as an AFS.

Example 2.5 (map as AFS)**.** The signature $\mathcal{F}$ consists of

$$\begin{aligned} \mathsf{0} &: \mathsf{nat} \\ \mathsf{s} &: \mathsf{nat} \to \mathsf{nat} \\ \mathsf{nil} &: \mathsf{natlist} \\ \mathsf{cons} &: (\mathsf{nat} \times \mathsf{natlist}) \to \mathsf{natlist} \\ \mathsf{map} &: ((\mathsf{nat} \to \mathsf{nat}) \times \mathsf{natlist}) \to \mathsf{natlist} \end{aligned}$$

There are two rewrite rules in $\mathcal{R}$:

$$\begin{aligned} \mathsf{map}(F, \mathsf{nil}) &\to \mathsf{nil} \\ \mathsf{map}(F, \mathsf{cons}(h, t)) &\to \mathsf{cons}(F \cdot h, \mathsf{map}(F, t)) \end{aligned}$$

Terms are rewritten using rewrite rules and β-reduction.

Definition 2.6. For a rule $l \to r \in \mathcal{R}$, a substitution γ and a context C, the rewrite relation $\to_{\mathcal{R}}$ is defined by

$$\begin{aligned} &\text{(rule)} \; C[l\gamma] \to_{\mathcal{R}} C[r\gamma] \\ &\text{(beta)} \; C[\lambda x : \sigma.s \cdot t] \to_{\mathcal{R}} C[s[x : \sigma := t]] \end{aligned}$$

Sometimes $\to_\beta$ is used for a rewrite step for clarification, indicating that (beta) was used.

The next example shows a rewrite sequence in the AFS from Example 2.5.

Example 2.7. Mapping the identity function, $\lambda x.x$, on the list containing the single element $\mathsf{s}(\mathsf{0})$ yields the following derivation:

$$\begin{aligned} & \underline{\mathsf{map}(\lambda x.x, \mathsf{cons}(\mathsf{s}(\mathsf{0}), \mathsf{nil}))} \\ \to_{\mathcal{R}} \; & \mathsf{cons}(\underline{(\lambda x.x) \cdot \mathsf{s}(\mathsf{0})}, \mathsf{map}(\lambda x.x, \mathsf{nil})) \\ \to_{\beta} \; & \mathsf{cons}(\mathsf{s}(\mathsf{0}), \underline{\mathsf{map}(\lambda x.x, \mathsf{nil})}) \\ \to_{\mathcal{R}} \; & \mathsf{cons}(\mathsf{s}(\mathsf{0}), \mathsf{nil}) \end{aligned}$$

2.3 Higher-Order Rewrite Systems

The second variant we consider are the Higher-Order Rewrite Systems by Nipkow [18]. Here $\lambda^{\to}$ is extended by typed constants. Terms are in η-long β-normal form and rewriting uses higher-order pattern matching.

Definition 2.8. A *signature* is a set of typed constants $\mathcal{F}$. A *pre-term* is an expression s, such that $s : \sigma$ for some type σ can be inferred using (var), (app), (abs) and

$$\text{(fun)} \; f : \sigma \quad \text{if } f : \sigma \in \mathcal{F}$$

A *term* is pre-term in η-long β-normal form, i.e., every pre-term s corresponds to a unique term $s\updownarrow_\beta^\eta$.[1]

Definition 2.9. A *rewrite rule* over $\mathcal{F}$ and $\mathcal{V}$ is a pair of terms $l \to r$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$ such that l and r have the same base type and all free variables of r occur in l. A HRS consists of a signature $\mathcal{F}$ and a set $\mathcal{R}$ of rules.

For a rule $l \to r \in \mathcal{R}$, a substitution γ and a context C, the rewrite relation $\to_\mathcal{R}$ is defined by

$$C[l\gamma\updownarrow_\beta^\eta] \to_\mathcal{R} C[r\gamma\updownarrow_\beta^\eta]$$

Unfortunately this rewrite relation is not decidable in general. Thus attention is commonly restricted to so called Pattern Higher-Order Rewrite Systems (PRS).

Definition 2.10. A term s is called a *pattern*, if for all subterms of s of the form $x \cdot t_1 \cdots t_n$ with $x \in \mathsf{FV}(s)$ and $n > 0$, all t_i are the η-long forms of different bound variables. A HRS $\mathcal{R}$ is a Pattern HRS if for all $l \to r \in \mathcal{R}$, l is a pattern.

For example $\lambda x.c \cdot x$, F, $\lambda x.F \cdot (\lambda z.x \cdot z)$ and $\lambda xy.F \cdot x \cdot y$ are patterns, while $F \cdot c$, $\lambda x.F \cdot x \cdot x$, $\lambda xy.F \cdot y \cdot c$ and $\lambda x.G \cdot (H \cdot x)$ are not. The main result about patterns is due to Miller [17] and states that unification is decidable for patterns and that if they are unifiable, a most general unifier can be computed.

Example 1.1 from the introduction can also be modelled as a PRS.

Example 2.11 (map as PRS)**.** The signature $\mathcal{F}$ consists of

$$\begin{aligned}
\mathsf{0} &: \mathsf{nat}\\
\mathsf{s} &: \mathsf{nat} \to \mathsf{nat}\\
\mathsf{nil} &: \mathsf{natlist}\\
\mathsf{cons} &: \mathsf{nat} \to \mathsf{natlist} \to \mathsf{natlist}\\
\mathsf{map} &: (\mathsf{nat} \to \mathsf{nat}) \to \mathsf{natlist} \to \mathsf{natlist}
\end{aligned}$$

The two rewrite rules in $\mathcal{R}$ are:

$$\begin{aligned}
\mathsf{map} \cdot (\lambda x.F \cdot x) \cdot \mathsf{nil} &\to \mathsf{nil}\\
\mathsf{map} \cdot (\lambda x.F \cdot x) \cdot (\mathsf{cons} \cdot h \cdot t) &\to \mathsf{cons} \cdot (F \cdot h) \cdot (\mathsf{map} \cdot (\lambda x.F \cdot x) \cdot t)
\end{aligned}$$

The computation from before performed in this PRS looks as follows:

$$\begin{aligned}
& \underline{\mathsf{map} \cdot (\lambda x.x) \cdot (\mathsf{cons} \cdot (\mathsf{s} \cdot \mathsf{0}) \cdot \mathsf{nil})}\\
\to_\mathcal{R}\ & (\mathsf{cons} \cdot (\underline{(\lambda x.x) \cdot (\mathsf{s} \cdot \mathsf{0})}) \cdot (\mathsf{map} \cdot (\lambda x.x) \cdot \mathsf{nil}))\updownarrow_\beta^\eta\\
=\ & \mathsf{cons} \cdot (\mathsf{s} \cdot \mathsf{0}) \cdot (\underline{\mathsf{map} \cdot (\lambda x.x) \cdot \mathsf{nil}})\\
\to_\mathcal{R}\ & \mathsf{cons} \cdot (\mathsf{s} \cdot \mathsf{0}) \cdot \underline{(\mathsf{nil}\updownarrow_\beta^\eta)}\\
=\ & \mathsf{cons} \cdot (\mathsf{s} \cdot \mathsf{0}) \cdot \mathsf{nil}
\end{aligned}$$

[1] In the literature terms $\lambda x_1 \ldots x_n.a \cdot s_1 \cdots s_n$ are often written as $\lambda x_1 \ldots x_n.a(s_1, \ldots, s_n)$. Since this notation disguises the fact that a is actually a constant and to avoid confusion with AFSs we will not adopt this convention here.

Note that in the first step the term does match the left-hand side of the second rule because $(\lambda y.(\lambda x.x) \cdot y) \to_\beta (\lambda y.y) \equiv_\alpha (\lambda x.x)$ and so

$$(\mathsf{map} \cdot (\lambda x.F \cdot x) \cdot (\mathsf{cons} \cdot h \cdot t))\gamma\downarrow^\eta_\beta = \mathsf{map} \cdot (\lambda x.x) \cdot (\mathsf{cons} \cdot (\mathsf{s} \cdot \mathsf{0}) \cdot \mathsf{nil})$$

with $\gamma = [F := \lambda x.x, h := \mathsf{s} \cdot \mathsf{0}, t := \mathsf{nil}]$.

3 Dependency Pairs

A first definition of higher-order dependency pairs is given by Sakai, Watanabe and Sakabe [20] in what is now called the dynamic style. The main drawback of their definition is that an ordering used to show absence of infinite chains must have the subterm-property: $C[s] \geq s$ for all terms s and contexts C. This makes optimisations like argument filterings impossible. The static approach does not require the subterm property, but is only applicable to a limited class of systems.

3.1 First-Order Dependency Pairs

Before introducing higher-order DPs we will recall and discuss the main ideas and definitions from the first-order setting. First defined in [1] dependency pairs have been continuously refined—state of the art first-order termination tools feature the dependency pair framework [6], where sets of DPs are iteratively transformed by dependency pair processors. Here we will use a version close to the higher-order definitions that are introduced afterwards.

The intuition of dependency pairs is to identify those parts of right-hand sides of rules that may give rise to a non-terminating rewrite sequence—intuitively the "recursive calls" are analysed. Then the main idea is to consider a minimally non-terminating term, i.e., a term all of whose proper subterms are terminating. In an infinite reduction starting at such a term eventually a step at the root, $l\gamma \to r\gamma$, must occur. Now one finds, that in a minimally non-terminating subterm of $r\gamma$ the root has to be a defined symbol, which already occurs in r. Thus one focuses on subterms of right-hand sides which have a defined symbol at the root.

Definition 3.1. Let $(\mathcal{F}, \mathcal{R})$ be a first-order TRS. A *defined symbol* is a function symbol, which occurs at the root of the left-hand side of a rewrite rule; we collect all defined symbols of $\mathcal{R}$ in $\mathcal{D}$. By $\mathcal{F}^\#$ we denote the set $\mathcal{F} \cup \{f^\# \mid f \in \mathcal{F}\}$, where $f^\#$ is a marked version of f with the same arity. Marking a term s is defined as $s^\# = f^\#(s_1, \ldots, s_n)$ if $s = f(s_1, \ldots s_n)$ and $f \in \mathcal{D}$ and $s^\# = s$ otherwise. The set of *candidate terms* of a term r is $\mathsf{Cand}(r) = \{p \mid r \trianglerighteq p \text{ and } \mathsf{root}(r) \in \mathcal{D}\}$. Then the set of *dependency pairs* of $\mathcal{R}$ is defined as $\mathsf{DP}(\mathcal{R}) = \{l^\# \rightsquigarrow p^\# \mid l \to r \in \mathcal{R} \text{ and } p \in \mathsf{Cand}(r)\}$.

An infinite reduction as described above then corresponds to an infinite *dependency chain*—a sequence $[(l_i \rightsquigarrow p_i, s_i, t_i) \mid i \in \mathbb{N}]$ such that for all i:

1. $l_i \rightsquigarrow p_i \in \mathsf{DP}(\mathcal{R})$,

2. $s_i = l_i\gamma$ and $t_i = p_i\gamma_i$ for some substitution γ_i,

3. $t_i \to_{\mathcal{R}}^* s_{i+1}$

Note that, since t_i is of the shape $f^\#(s_1, \ldots, s_n)$ and $f^\#$ does not occur in $\mathcal{R}$, the reduction in (3) takes place below the root. The following is the main result about dependency pairs.

Theorem 3.2 (Arts and Giesl, 2000)**.** *A TRS is terminating if and only if it does not admit an infinite dependency chain.*

Now the challenge is to show absence of infinite chains.

Definition 3.3. A *reduction pair* is a pair $(>,\geq)$ consisting of a well-founded ordering $>$ and a quasi-ordering $\geq$, such that:

- $>$ and $\geq$ are compatible, i.e., either $> \cdot \geq \subseteq >$ or $\geq \cdot > \subseteq >$
- both $>$ and $\geq$ are stable, i.e., closed under substitution
- $\geq$ is monotone, i.e., closed under contexts

The main advantage is that $>$ need not be monotone. Intuitively this means that arguments can be discarded by the ordering. This can be exploited by using, for example, non-monotone polynomial interpretations or by adapting an ordering like RPO using argument filterings. The following result shows how reduction pairs can be used to show termination.

Theorem 3.4. *Let $(>,\geq)$ be a reduction pair such that $l > p$ for all $l \rightsquigarrow p \in \mathsf{DP}(\mathcal{R})$ and $l \geq r$ for all $l \to r \in \mathcal{R}$. Then $\mathcal{R}$ is terminating.*

When trying to lift these definitions to higher-order formalisms several problems occur.

Collapsing rules A rule $l \to r$ is called collapsing if $\mathsf{head}(r)$ is a variable. Collapsing rules may lead to non-termination, which is not captured by dependency pairs that only consider defined symbols.

Example 3.5. Consider the AFS consisting of the single rule $\mathsf{f}(\mathsf{g}(y), x) \to y \cdot x$. Here the right-hand side does not even contain a function symbol. Yet an infinite reduction exists:

$$\begin{aligned} & \mathsf{f}(\mathsf{g}(\lambda x.\mathsf{f}(x,x)), \mathsf{g}(\lambda x, \mathsf{f}(x,x))) \\ \to_{\mathcal{R}}\ & (\lambda x.\mathsf{f}(x,x)) \cdot \mathsf{g}(\lambda x, \mathsf{f}(x,x)) \\ \to_{\beta}\ & \mathsf{f}(\mathsf{g}(\lambda x.\mathsf{f}(x,x)), \mathsf{g}(\lambda x, \mathsf{f}(x,x))) \\ \to_{\mathcal{R}}\ & \cdots \end{aligned}$$

Rules of functional type A similar problem occurs if rules of functional type are present in an AFS (in a HRS rules are demanded to have base type).

Example 3.6. Consider the AFS containing the rule $\mathsf{f}(\mathsf{g}(x)) \to x$. This harmless looking rule can cause non-termination if x is higher-order variable, i.e., if we have type declarations like $f : \mathsf{o} \to \mathsf{o} \to \mathsf{o}$ and $g : (\mathsf{o} \to \mathsf{o}) \to \mathsf{o}$. Then we have the following infinite reduction:

$$\begin{aligned} & \mathsf{f}(\mathsf{g}(\lambda x.\mathsf{f}(x) \cdot x)) \cdot \mathsf{g}(\lambda x.\mathsf{f}(x) \cdot x) \\ \to_{\mathcal{R}} & (\lambda x.\mathsf{f}(x) \cdot x) \cdot \mathsf{g}(\lambda x.\mathsf{f}(x) \cdot x) \\ \to_{\beta} & \mathsf{f}(\mathsf{g}(\lambda x.\mathsf{f}(x) \cdot x)) \cdot \mathsf{g}(\lambda x.\mathsf{f}(x) \cdot x) \\ \to_{\mathcal{R}} & \cdots \end{aligned}$$

Dangling variables When just taking subterms of right-hand sides in dependency pairs, bound variables might become free leading to infinite dependency chains, although the rewrite system is terminating.

Example 3.7. Consider a rule $\mathsf{f}(\mathsf{0}) \to \mathsf{g}(\lambda x.\mathsf{f}(x))$. Applying a naive dependency pair definition might result in a pair $\mathsf{f}^{\#}(\mathsf{0}) \rightsquigarrow \mathsf{f}^{\#}(x)$, which obviously gives rise to an infinite chain.

Typing issues Since in dependency pairs we take subterms of right-hand sides of rewrite rules, in a DP $l \rightsquigarrow p$, l and p need not have the same type. However orderings like HORPO are only equipped to compare terms of the same type (modulo renaming of base types).

3.2 Static Higher-Order Dependency Pairs for HRSs

The first variant of dependency pairs we discuss are usually called static dependency pairs and are commonly defined for HRSs (or PRSs). The definition is close to that of first-order dependency pairs and hence we present them first—however they can only be applied to a restricted class of systems. Here we discuss the dependency pairs proposed by Sakai and Kusakari [19], which are defined for PRSs.

First we adapt the definition of defined symbols. The concept of root is now played by the *top* of a term.

Definition 3.8. Let s be a term with respect to Definition 2.8. Then s can be written in the form $\lambda x_1 \dots x_n.\, a \cdot s_1 \cdots s_n$ and the top of s is defined as $\mathsf{top}(s) = a$.[2] The set of defined symbols with respect to a PRS $\mathcal{R}$ is $\mathcal{D} = \{\mathsf{top}(l) \mid l \to r \in \mathcal{R}\}$.

Now dependency pairs are defined just like in the first-order case.

Definition 3.9. Let r be a term. the candidate terms of r are

$$\mathsf{Cand}(r) = \{p \mid r \trianglerighteq p \text{ and } \mathsf{top}(p) \in \mathcal{D}\}$$

[2] So the top of a term is the head after stripping away abstractions.

We again extend the signature by marked versions of all functions symbols and define $s^\# = f^\# \cdot t_1 \cdots t_n$ if $s = f \cdot t_1 \cdots t_n$ and $f \in \mathcal{D}$; $s^\# = s$ otherwise. Let $\mathcal{R}$ be a PRS. The set of dependency pairs of $\mathcal{R}$ is defined as

$$\mathsf{DP}(\mathcal{R}) = \{l^\# \rightsquigarrow p^\# \mid l \to r \in \mathcal{R} \text{ and } p \in \mathsf{Cand}(r)\}$$

Example 3.10. Consider the PRS from Example 2.11. We have one static dependency pair:

$$\mathsf{map}^\# \cdot (\lambda x.F \cdot x) \cdot (\mathsf{cons} \cdot h \cdot t) \rightsquigarrow \mathsf{map}^\# \cdot (\lambda x.F \cdot x) \cdot t$$

Definition 3.11. Let $\mathcal{R}$ be a PRS. A sequence $[(l_i \rightsquigarrow p_i, s_i, t_i) \mid i \in \mathbb{N}]$ is called dependency chain if

1. $l_i \rightsquigarrow p_i \in \mathsf{DP}(\mathcal{R})$
2. $s_i = l_i\gamma_i$ and $t_i = p_i\gamma_i$ for some substitution γ_i
3. $t_i \mathrel{\updownarrow^\eta_\beta} \to^*_\mathcal{R} s_{i+1} \mathrel{\updownarrow^\eta_\beta}$

These definitions are very close to their first-order counter-parts. However, they induce several limitations and problems that will be discussed in the following.

First, to show soundness it is necessary to construct an infinite dependency chain from an infinite reduction. However the method from the first-order case is not applicable.

Example 3.12. Consider a PRS $\mathcal{R}$ consisting of the following three rules:

$$\mathsf{f} \cdot (\lambda x.F \cdot x) \to F \cdot \mathsf{a} \qquad \mathsf{g} \cdot \mathsf{a} \to \mathsf{f} \cdot (\lambda x.\mathsf{i} \cdot x) \qquad \mathsf{i} \cdot x \to \mathsf{h} \cdot (\mathsf{g} \cdot x)$$

There are three dependency pairs in $\mathsf{DP}(\mathcal{R})$:

$$\mathsf{g}^\# \cdot \mathsf{a} \rightsquigarrow \mathsf{f}^\# \cdot (\lambda x.\mathsf{i} \cdot x) \qquad \mathsf{g}^\# \cdot \mathsf{a} \rightsquigarrow \mathsf{i}^\# \cdot x \qquad \mathsf{i}^\# \cdot x \rightsquigarrow \mathsf{g}^\# \cdot x$$

We have an infinite reduction

$$\mathsf{g} \cdot \mathsf{a} \to_\mathcal{R} \mathsf{f} \cdot (\lambda x.\mathsf{i} \cdot \mathsf{a}) \to_\mathcal{R} \mathsf{i} \cdot \mathsf{a} \to_\mathcal{R} \mathsf{h} \cdot (\mathsf{g} \cdot \mathsf{a}) \to_\mathcal{R} \ldots$$

and an infinite dependency chain

$$\begin{aligned}[]
&[(\mathsf{g}^\# \cdot \mathsf{a} \rightsquigarrow \mathsf{i}^\# \cdot x, \mathsf{g}^\# \cdot \mathsf{a}, \mathsf{i}^\# \cdot \mathsf{a}),\\
&\ (\mathsf{i}^\# \cdot x \rightsquigarrow \mathsf{g}^\# \cdot x, \mathsf{i}^\# \cdot \mathsf{a}, \mathsf{g}^\# \cdot \mathsf{a}),\\
&\ (\mathsf{g}^\# \cdot \mathsf{a} \rightsquigarrow \mathsf{i}^\# \cdot x, \mathsf{g}^\# \cdot \mathsf{a}, \mathsf{i}^\# \cdot \mathsf{a}),\\
&\ \ldots]
\end{aligned}$$

The dependency chain in the above example completely ignores the root step using the first rewrite rule. Obviously this has to be the case, since there are no DPs originating from that rule. Thus a slight modification of the above example also immediately shows that the method is not complete. Just drop the first rule, then the same infinite chain exists, but there is no infinite reduction

anymore. The reason is of course that the bound variable x in the right-hand side of the second rule became free in a dependency pair.

Not having completeness is the accepted price to pay for static dependency pairs. To show soundness, Sakai and Kusakari take the following approach. From a minimal non-terminating sequence they construct a *dependency forest*, which is basically a graph keeping track of the descendants in a infinite reduction. Then from an infinite path in a dependency forest they construct an infinite chain. The main idea then is:

- If there is an infinite reduction the dependency forest has infinitely many nodes.
- If additionally the dependency forest is finitely branching there has to be an infinite path.
- Hence there is an infinite dependency chain.

Thus non-existence of an infinite dependency chain implies termination of a PRS $\mathcal{R}$, provided dependency forests for $\mathcal{R}$ are finitely branching. See [19] for details on dependency forests. The authors then continue by giving sufficient conditions for a PRS to have dependency forests which are finitely branching.

Definition 3.13. A term s is *strongly linear* if it has an α-equal term in which every variable (bound or free) occurs only once. A PRS is strongly linear if all right-hand sides of rewrite rules are strongly linear.

For example $x \cdot y$ and $(\lambda x.x) \cdot (\lambda x.x)$ are strongly linear, $\mathsf{f} \cdot x \cdot x$ and $\lambda x.\mathsf{f} \cdot x \cdot x$ are not.

Definition 3.14. A term s is *nested with respect* to a set of variables $V \subseteq \mathcal{V}$ if it contains a subterm $x \cdot s_1 \cdots s_n$ such that $x \in V$ and

- $V \cap \mathsf{FV}(s_i) \neq \varnothing$ for some $1 \leqslant i \leqslant n$ or
- $s_i = \lambda x_1 \ldots x_m.\, s'$ and s' is nested with respect to $\{x_1, \ldots, x_m\}$ for some $1 \leqslant i \leqslant n$

A term s is *nested* if it is nested with respect to $\mathsf{FV}(s)$. A HRS $\mathcal{R}$ is *non-nested* if r is not nested for every $l \to r \in R$.

For instance $x \cdot y$ and $x \cdot (\lambda yz.y \cdot z)$ are nested, while $\mathsf{f} \cdot (\lambda yz.y \cdot z)$ and $\mathsf{f} \cdot (\lambda x.y \cdot y)$ are not. Forbidding nested right-hand sides basically forbids collapsing rules that might cause non-termination. The following then the main result.

Lemma 3.15 (Sakai and Kusakari, 2005)**.** *Let $\mathcal{R}$ be a PRS and DF be a dependency forest for a minimal non-terminating sequence A.*

1. *If $\mathcal{R}$ is strongly linear and A begins at a strongly linear term, then DF is finitely branching*
2. *If $\mathcal{R}$ is non-nested then DF is finitely branching.*

Corollary 3.16. *Let $\mathcal{R}$ be a PRS, which does not admit an infinite dependency chain. If $\mathcal{R}$ is strongly linear, then every strongly linear term is terminating. If $\mathcal{R}$ is non-nested then $\mathcal{R}$ is terminating.*

Example 3.17. The AFS from Example 3.5 can also be modelled as a PRS $\mathcal{R}$, which is nested and contains the following rule:

$$\mathsf{f} \cdot (\mathsf{g} \cdot x) \cdot y \to x \cdot y$$

There is no dependency pair, yet we have the following infinite reduction:

$$\begin{aligned} & \mathsf{f} \cdot (g \cdot (\lambda x.\mathsf{f} \cdot x \cdot x)) \cdot (g \cdot (\lambda x.\mathsf{f} \cdot x \cdot x)) \\ \to_{\mathcal{R}} & ((\lambda x.f \cdot x \cdot x) \cdot (g \cdot (\lambda x.\mathsf{f} \cdot x \cdot x))) \updownarrow_{\beta}^{\eta} \\ = \quad & \mathsf{f} \cdot (g \cdot (\lambda x.\mathsf{f} \cdot x \cdot x)) \cdot (g \cdot (\lambda x.\mathsf{f} \cdot x \cdot x)) \\ \to_{\mathcal{R}} & \cdots \end{aligned}$$

However, $\mathcal{R}$ is strongly linear and hence every strongly linear term is terminating. Indeed the infinite reduction above starts at a term, which is not strongly linear.

Another possible class of systems where static dependency pairs can be applied is that of *plain function passing* HRSs, see [16]. Moreover argument filterings and usable rules have been defined, see [21]. We now turn to the dynamic style of dependency pairs.

3.3 Dynamic Higher-Order Dependency-Pairs for AFSs

Originally the first approach but then rather neglected, dynamic dependency pairs recently resurfaced, when Kop and van Raamsdonk defined them for Algebraic Functional Systems [15]. The main differences to the static dependency pairs above are, that also subterms headed by a free variable will lead to a dependency pair and that bound variables will not become free.

First, several pre-processing steps are made to handle functional rules and dangling variables. Using results from [14] it is assumed that all rules $l \to r$ of an AFS fulfil the following restrictions:

- l and r do not contain subterms of the form $(\lambda x.s) \cdot t$, i.e., β-redexes
- l does not contain a subterm of the form $x \cdot s$ with $x \in \mathsf{FV}(l)$
- l has the shape $f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m$ with $m \geqslant n \geqslant 0$.

Then the set of defined symbols for an AFS $\mathcal{R}$ is defined as

$$\mathcal{D} = \{f \mid f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m \to r \in \mathcal{R}\}$$

As before we use marked versions of function symbols. Marking a term is now defined as $s^{\#} = f^{\#}(s_1, \ldots, s_n)$ if $s = f(s_1, \ldots, s_n)$ and $f \in \mathcal{D}$ and $s^{\#} = s$ otherwise. In particular applications are not marked: $(f(x) \cdot y)^{\#} = f(x) \cdot y$. Moreover to handle certain rules of functional type a pre-processing step called *completion* is employed.

Definition 3.18. An AFS $\mathcal{R}$ is completed by adding for each rule of the shape $l \to \lambda x_1 \dots x_n.\, r$, where r is not an abstraction, the following n new rules:

$$l \cdot x_1 \to \lambda x_2 \dots x_n.\, r,\ \dots,\ l \cdot x_1 \cdots x_n \to r$$

Example 3.19. An example where completion is necessary is the AFS consisting of the rule $\mathsf{f}(\mathsf{a}) \to \lambda x.\mathsf{f}(x) \cdot x$. There is an infinite reduction using this rule:

$$\mathsf{f}(\mathsf{a}) \cdot \mathsf{a} \to_{\mathcal{R}} (\lambda x.\mathsf{f}(x) \cdot x) \cdot \mathsf{a} \to_{\beta} \mathsf{f}(\mathsf{a}) \cdot \mathsf{a} \to_{\mathcal{R}} \dots$$

However the term $\mathsf{f}(\mathsf{a})$ itself is terminating. The problem is that the rule has functional type. Completion makes sure that the important reduction step could also be taken at the root, by adding the following rule of basic type: $\mathsf{f}(\mathsf{a}) \cdot x \to \mathsf{f}(x) \cdot x$.

However, not all rules with functional type are completed, only those where the right hand side is an abstraction. The others are handled directly in the definition of dependency pairs.

When defining candidate terms for dependency pairs it is made sure that bound variables do not become free by taking subterms. This is achieved by replacing them by fresh constants. Thus $F \cdot c$ will be a subterm of $\lambda x.F \cdot (F \cdot x)$. To this end a set $\mathcal{C}$, which contains for every variable x a fresh constant symbol c_x of the same type as x.

Definition 3.20. Let s be a term. Then a term $t[x_1 := c_{x_1}, \dots, x_n := c_{x_n}]$ is a *candidate term* of s if $s \trianglerighteq t$, $\mathsf{BV}(s) \cap \mathsf{FV}(t) = \{x_1, \dots, x_n\}$ and either

1. $t = f(t_1, \dots, t_n) \cdot t_{n+1} \cdots t_m$ with $m \geqslant n \geqslant 0$ and $f \in \mathcal{D}$ or
2. $t = x \cdot t_1 \cdots t_n$ with $n > 0$ and $x \in \mathsf{FV}(s)$.

We collect all candidate terms of s in $\mathsf{Cand}(s)$.

Clause 1 is the expected case for defined symbols, whereas Clause 2 (partly) handles subterms headed by higher-order variables. The substitution $[x_1 := c_{x_1}, \dots, x_n := c_{x_n}]$ ensures that variables which occur bound in s but free in the subterm t are replaced by fresh constants in the candidate term.

Definition 3.21. Let $\mathcal{R}$ be an AFS. The set of dependency pairs of a rule $l \to r \in \mathcal{R}$, $\mathsf{DP}(l \to r)$, contains:

1. all pairs $l^\# \rightsquigarrow p^\#$ with $p \in \mathsf{Cand}(r)$ and
2. all pairs $l \cdot x_1 \cdots x_k \rightsquigarrow r \cdot x_1 \cdots x_k$, $1 \leqslant k \leqslant n$ with
   - $r : \sigma_1 \to \dots \sigma_n \to \iota$, i.e., r is of functional type and either
   - $r = x \cdot r_1 \cdots r_m$ with $m \geqslant 0$ or
   - $r = f(r_1, \dots r_i) \cdot r_{i+1} \cdots r_m$ with $m \geqslant i \geqslant 0$ and $f \in \mathcal{D}$

Here the second case handles rules of functional type, which have not been completed when pre-processing the rules.

Example 3.22. For example the rule $\mathsf{f}(\mathsf{0}) \to \mathsf{g}(\lambda x.\mathsf{f}(x))$ now generates a dependency pair $\mathsf{f}^\#(\mathsf{0}) \rightsquigarrow \mathsf{f}^\#(c_x)$.

The rule $\mathsf{f}(\mathsf{g}(F), x) \to F \cdot x$ generates a dependency pair $\mathsf{f}^\#(g(F), x) \rightsquigarrow F \cdot x$, because $F \cdot x \in \mathsf{Cand}(F \cdot x)$ by Clause 2 of Definition 3.20.

The rule $\mathsf{f}(\mathsf{g}(x)) \to x$ generates a dependency pair $\mathsf{f}(\mathsf{g}(x)) \cdot x_1 \to x \cdot x_1$ by Clause 2 of Definition 3.21.

Example 3.23. The AFS from Example 2.5 admits the following two dynamic dependency pairs.

$$\mathsf{map}^\#(F, \mathsf{cons}(h, t)) \rightsquigarrow F \cdot h \qquad \mathsf{map}^\#(F, \mathsf{cons}(h, t)) \rightsquigarrow \mathsf{map}^\#(F, t)$$

The definition of dependency chain has to be adapted to take into account that β-reduction is available as explicit rewrite step.

Definition 3.24. A dependency chain for an AFS $\mathcal{R}$ is a sequence $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ such that for all i

1. $\rho_i \in \mathsf{DP}(\mathcal{R}) \cup \{\mathsf{beta}\}$.
2. if $\rho_i = l_i \rightsquigarrow p_i \in \mathsf{DP}(\mathcal{R})$ then there is a substitution γ_i with $s_i = l_i\gamma_i$ and $t_i = p_i\gamma_i$.
3. if $\rho_i = \mathsf{beta}$ then $s_i = (\lambda x.u) \cdot v \cdot w_1 \cdots w_k$ and either
   - $k > 0$ and $t_i = u[x := v] \cdot w_1 \cdots w_k$ or
   - $k = 0$ and there is a term w with $u \trianglerighteq w$ and $x \in \mathsf{FV}(w)$ and $w \neq x$ and $t_i = w^\#[x := v]$
4. $t_i \xrightarrow{in}{}^*_{\mathcal{R}} s_{i+1}$

Here $\xrightarrow{in}_{\mathcal{R}}$ denotes rewriting inside a term of the shape $f(u_1, \ldots, u_n) \cdot u_{n+1} \cdots u_m$ a term u_i, $1 \leqslant i \leqslant m$.

The first clause states we either make a dependency pair step or beta step. The second clause is the expected definition for dependency pair steps. The third clause handles beta steps and deserves further explanation. The fourth clause allows rewriting inside the terms between topmost steps—this condition has to be made explicit, because t_i might be an application and applications are not marked. For beta steps the first subcase ($k > 0$) is straight-forward, it just corresponds to the definition of β-reduction. However the second subcase ($k = 0$) is more involved—in particular one has to allow using a subterm w of u for the β-step. After the β-step the result has to be marked, since it might be a functional term, to allow potential further dependency pair steps. To see why allowing a subterm is necessary consider the following example. Intuitively the reason is that β-steps might wrap the minimally non-terminating term, where the chain should continue, into a context.

Example 3.25. Let $\mathcal{R}$ be the AFS consisting of the single rule $\mathsf{f}(\mathsf{g}(F)) \to F \cdot \mathsf{g}(F)$. We have one dependency pair $\mathsf{f}^\#(\mathsf{g}(F)) \rightsquigarrow F \cdot \mathsf{g}(F)$. Now consider

the minimally non-terminating term $\mathsf{f}(\mathsf{g}(\lambda x.\mathsf{h}(\mathsf{f}(x))))$, which gives rise to the infinite reduction

$$\mathsf{f}(\mathsf{g}(\lambda x.\mathsf{h}(\mathsf{f}(x)))) \to_{\mathcal{R}} (\lambda x.\mathsf{h}(\mathsf{f}(x))) \cdot \mathsf{g}(\lambda x.\mathsf{h}(\mathsf{f}(x))) \to_{\beta} \mathsf{h}(\mathsf{f}(\mathsf{g}(\lambda x.\mathsf{h}(\mathsf{f}(x))))) \to_{\mathcal{R}} \dots$$

The corresponding infinite dependency chain is:

$$\begin{aligned}&[(\mathsf{f}^{\#}(\mathsf{g}(F)) \rightsquigarrow F \cdot \mathsf{g}(F), \mathsf{f}^{\#}(\mathsf{g}(\lambda x.\mathsf{h}(\mathsf{f}(x)))), (\lambda x.\mathsf{h}(\mathsf{f}(x))) \cdot \mathsf{g}(\lambda x.\mathsf{h}(\mathsf{f}(x)))),\\ &(\mathsf{beta}, (\lambda x.\mathsf{h}(\mathsf{f}(x))) \cdot \mathsf{g}(\lambda x.\mathsf{h}(\mathsf{f}(x))), \mathsf{f}^{\#}(\mathsf{g}(\lambda x.\mathsf{h}(\mathsf{f}(x))))),\\ &\dots]\end{aligned}$$

Here in the second step we needed to take $\mathsf{h}(\mathsf{f}(x)) \trianglerighteq \mathsf{f}(x)$, i.e., $u = \mathsf{h}(\mathsf{f}(x))$ and $w = \mathsf{f}(x)$. Because otherwise we would have ended up with $\mathsf{h}^{\#}(\mathsf{f}(\mathsf{g}(\lambda x.\mathsf{h}(\mathsf{f}(x)))))$, where the chain could not have been continued. This example also shows the need to mark w after doing a beta step at the top: there would have been no possibility to continue the chain from $\mathsf{f}(\mathsf{g}(\lambda x.\mathsf{h}(\mathsf{f}(x))))$.

The following are the main results of [15].

Theorem 3.26 (Kop and van Raamsdonk, 2011)**.** *Let $\mathcal{R}$ be an AFS. If it is non-terminating then there is an infinite dependency chain over $\mathsf{DP}(\mathcal{R})$.*

For left-linear AFSs the absence of infinite chains fully characterises termination.

Theorem 3.27 (Kop and van Raamsdonk, 2011)**.** *A left-linear AFS $\mathcal{R}$ is terminating if and only if it does not admit and infinite dependency chain.*

To see why non left-linearity can cause trouble consider the following AFS from [15].

Example 3.28. Consider the AFS $\mathcal{R}$ consisting of the following two rules:

$$\mathsf{f}(x, y, \mathsf{s}(z)) \to \mathsf{g}(\mathsf{h}(x, y), \lambda u.\mathsf{f}(u, x, z)) \qquad \mathsf{h}(x, x) \to \mathsf{f}(x, \mathsf{s}(x), \mathsf{s}(\mathsf{s}(x)))$$

We have three dependency pairs in $\mathsf{DP}(\mathcal{R})$:

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

There is an infinite dependency chain:

$$\begin{aligned}&[(\mathsf{f}^{\#}(x, y, \mathsf{s}(z)) \rightsquigarrow \mathsf{f}^{\#}(c_u, x, z), \mathsf{f}^{\#}(c_u, \mathsf{s}(c_u), \mathsf{s}(\mathsf{s}(c_u))), \mathsf{f}^{\#}(c_u, c_u, \mathsf{s}(c_u))),\\ &(\mathsf{f}^{\#}(x, y, \mathsf{s}(z)) \rightsquigarrow \mathsf{h}^{\#}(x, y), \mathsf{f}^{\#}(c_u, c_u, \mathsf{s}(c_u)), \mathsf{h}^{\#}(c_u, c_u)),\\ &(\mathsf{h}^{\#}(x, x) \rightsquigarrow \mathsf{f}^{\#}(x, \mathsf{s}(x), \mathsf{s}(\mathsf{s}(x))), \mathsf{h}^{\#}(c_u, c_u), \mathsf{f}^{\#}(c_u, \mathsf{s}(c_u), \mathsf{s}(\mathsf{s}(c_u)))),\\ &\dots]\end{aligned}$$

Yet, the system is terminating. When trying to construct an infinite reduction, which corresponds to the infinite chain one gets:

$$\begin{aligned} & \mathsf{f}(c_u, \mathsf{s}(c_u), \mathsf{s}(\mathsf{s}(c_u))) \\ \to_{\mathcal{R}} & \mathsf{g}(\mathsf{h}(c_u, \mathsf{s}(c_u)), \lambda u.\mathsf{f}(u, c_u, \mathsf{s}(c_u))) \\ \to_{\mathcal{R}} & \mathsf{g}(\mathsf{h}(c_u, \mathsf{s}(c_u)), \lambda u.\mathsf{g}(\mathsf{h}(u, c_u), \lambda w.\mathsf{f}(w, u, c_u))) \end{aligned}$$

Now the bound variable u, that was replaced by c_u in the corresponding DP, is still there in $\mathsf{h}(u, c_u)$ and thus the second (non left-linear) rule is not applicable.

4 Proving Termination

Now, as in the first-order case the main challenge is to show absence of infinite dependency chains. As only a few other higher-order termination techniques exist, the choice of methods is rather limited (compared to first-order termination at least).

Static DPs To handle dependency pairs as introduced in Section 3.2 in [19] the following lemma is shown.

Lemma 4.1 (Sakai and Kusakari, 2005)**.** *Let $\mathcal{R}$ be a PRS. If there is a reduction pair $(>, \geq)$ such that*

1. *$l \geq r$ for all rules $l \to r \in R$ and*

2. *$l > p$ for all dependency pairs $l \rightsquigarrow p \in \mathsf{DP}(\mathcal{R})$*

then $\mathcal{R}$ has no infinite chain.

One basically has two reduction orderings available. On the one hand, one can use a HORPO variant. There a drawback is that the latest variant, the Computability Path Ordering [3], is only available for AFSs at the moment. The other possibility is a semantic approach due to van de Pol [22], which we will shortly discuss in the next section. On the plus side static dependency pairs allow argument filterings and usable rules, which have recently been defined [21].

Example 4.2. Consider the PRS $\mathcal{R}$ from Example 2.11, which admits the dependency pair $\mathsf{map}^{\#} \cdot (\lambda x.F \cdot x) \cdot (\mathsf{cons} \cdot h \cdot t) \rightsquigarrow \mathsf{map}^{\#} \cdot (\lambda x.F \cdot x) \cdot t$. Using a reduction pair $(\succ, \succeq)$, built from a HORPO version for HRSs, we have

$$\begin{aligned} \mathsf{map}^{\#} \cdot (\lambda x.F \cdot x) \cdot (\mathsf{cons} \cdot h \cdot t) &\succ \mathsf{map}^{\#} \cdot (\lambda x.F \cdot x) \cdot t \\ \mathsf{map} \cdot (\lambda x.F \cdot x) \cdot \mathsf{nil} &\succeq \mathsf{nil} \\ \mathsf{map} \cdot (\lambda x.F \cdot x) \cdot (\mathsf{cons} \cdot h \cdot t) &\succeq \mathsf{cons} \cdot (F \cdot h) \cdot (\mathsf{map} \cdot (\lambda x.F \cdot x) \cdot t) \end{aligned}$$

Thus $\mathcal{R}$ is terminating.

Dynamic DPs When dealing with the dependency pairs from Section 3.3 the main difficulty is to get rid of the subterm property requirement, which compromises the benefits of dependency pairs. Kop and van Raamsdonk propose the following approach.

Definition 4.3. A *reduction triple* $(>, \geq, \geq_1)$ 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
2. $>$, $\geq$ and $\geq_1$ are stable
3. $\geq_1$ is monotone
4. $\geq_1$ contains β, i.e., $(\lambda x.s) \cdot t \geq_1 s[x := t]$

A *reduction pair* is pair $(>, \geq)$ such that $(>, \geq, \geq)$ is a reduction triple.

An ordering $\geq$ 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 $(\lambda x.s) \cdot t \geq (u^{\#})\gamma[x := t]$.

Theorem 4.4 (Kop and van Raamsdonk, 2011)**.** *A set of dependency pairs $D = D_1 \uplus D_2$ does not admit an infinite chain if D_2 does not admit an infinite chain and there is a reduction triple $(>, \geq, \geq_1)$ such that*

1. *$l > p$ for all $l \rightsquigarrow p \in D_1$*
2. *$l \geq p$ for all $l \rightsquigarrow p \in D_2$*
3. *$l \geq_1 r$ for all $l \to r \in \mathcal{R}$*
4. *either D is non-collapsing, i.e., does not contain pairs where the right-hand side has the shape $x \cdot t_1 \cdots t_n$ or $\geq$ satisfies the limited subterm property.*

Kop and van Raamsdonk also define a dependency graph, which, like for first-order termination, allows to consider only strongly connected components. However, one still has to show $l \geq_1 r$ for all rules. Using a HORPO like ordering, this is hardly easier. Hence Kop and van Raamsdonk go on to adapt the semantic approach by van de Pol to AFSs.

4.1 Higher-Order Monotone Algebras

A semantic approach to show termination of first-order TRSs is to show that it is compatible with a well-founded monotone algebra. That is, terms are interpreted in a way, such that every rewrite step corresponds to a step in the well-founded order of the algebra. This method was extended by van de Pol to HRSs [22] and is further adapted in [15].

Here we only give the intuition and discuss the approach by means of an example. First every type σ is interpreted by a set of *weakly monotone functionals* WM_σ. Base types are interpreted by a carrier set A. A functional type $\sigma \to \tau$ is

interpreted by weakly monotone functions from WM_σ to WM_τ. Such functionals are written in lambda notation, for example $\boldsymbol{\lambda} n.n+1$ denotes the successor functional. Associating to every function symbol f a functional f_{WM}, terms can be interpreted. The application symbol gets a special treatment. Kop and van Raamsdonk basically interpret an application $s \cdot t$ as the functional computing the maximum of s applied to t and t. Additionally orderings $\sqsupseteq$ and $\sqsupset$ are defined by lifting the ordering $>$ on the carrier set to functionals. The main benefit is that $\sqsupset$ does not need to be monotone when using dependency pairs. See [15, 22] for further details

Example 4.5. The AFS $\mathcal{R}$ from Example 2.5 is terminating. To show this, we give a reduction pair $(\succ, \succeq)$ such that

$$\begin{aligned}
\mathsf{map}(F, \mathsf{nil}) &\succeq \mathsf{nil} \\
\mathsf{map}(F, \mathsf{cons}(h,t)) &\succeq \mathsf{cons}(F \cdot h, \mathsf{map}(F,t)) \\
\mathsf{map}^\#(F, \mathsf{cons}(h,t)) &\succ F \cdot h \\
\mathsf{map}^\#(F, \mathsf{cons}(h,t)) &\succ \mathsf{map}^\#(F,t)
\end{aligned}$$

We use an interpretation in the natural numbers. Consider the interpretation $\mathsf{map}^\#_{\mathsf{WM}} = \boldsymbol{\lambda} f.\boldsymbol{\lambda} x.f(x)+x$, $\mathsf{map}_{\mathsf{WM}} = \boldsymbol{\lambda} f.\boldsymbol{\lambda} x.x \times f(x)+x$, $\mathsf{cons}_{\mathsf{WM}} = \boldsymbol{\lambda} x.\boldsymbol{\lambda} y.x+y+1$ and $\mathsf{nil}_{\mathsf{WM}} = 0$. Evaluating the left- and right-hand sides of the rules under this interpretation we need to show

$$\begin{aligned}
0 \times F(0) + 0 &\geq 0 \\
(h+t+1) \times F(h+t+1) + h + t + 1 &\geq \max(F(h), h) + t \times F(t) + t + 1 \\
F(h+t+1) + h + t + 1 &> \max(F(h), h) \\
F(h+t+1) + h + t + 1 &> F(t) + t
\end{aligned}$$

for all natural numbers h, t and for all $F \in \mathsf{WM}_{\mathsf{nat}\to\mathsf{nat}}$, i.e., for all weakly monotone functions from natural numbers to natural numbers. The first constraint obviously holds. The third and the fourth constraint hold because F is weakly monotone and thus $F(h+t+1) \geq F(t)$ and $F(h+t+1) \geq F(h)$. For the second constraint we make a case distinction on the maximum. We need to show

$$\begin{aligned}
(h+t+1) \times F(h+t+1) + h + t + 1 &\geq h + t \times F(t) + t + 1 \\
(h+t+1) \times F(h+t+1) + h + t + 1 &\geq F(h) + t \times F(t) + t + 1
\end{aligned}$$

Expanding the left-hand side and simplifying yields

$$\begin{aligned}
h \times F(h+t+1) + t \times F(h+t+1) + F(h+t+1) &\geq t \times F(t) \\
h \times F(h+t+1) + t \times F(h+t+1) + F(h+t+1) + h &\geq F(h) + t \times F(t)
\end{aligned}$$

These hold because $t \times F(h+t+1) \geq t \times F(t)$ and $F(h+t+1) \geq F(h)$ by weak monotonicity.

4.2 Using First-Order Tools

A very different approach is proposed in [5], where Fuhs and Kop suggest how the power of first-order termination provers can be used to show termination of first-order functions in higher-order systems. The idea is that a higher-order termination tool might fail due to first-order rules it cannot handle. Yet such a rule might be easy for a powerful first-order termination tool. Thus splitting a system in a higher-order and a first-order part can be of advantage.

To this end Fuhs and Kop split the signature and the rewrite rules into two parts: the potentially higher-order symbols and rules ($\mathcal{F}_{\mathsf{PHO}}$, $\mathcal{R}_{\mathsf{PHO}}$) and the truly first-order symbols and rules ($\mathcal{F}_{\mathsf{TFO}}$, $\mathcal{R}_{\mathsf{TFO}}$). The main observation then is, that in a infinite chain, if the head of a minimally non-terminating term is in $\mathcal{F}_{\mathsf{TFO}}$, also the head of the next term in the chain is in $\mathcal{F}_{\mathsf{TFO}}$. Thus the following is the main result.

Theorem 4.6 (Fuhs and Kop, 2011)**.** *If $\mathcal{R}$ admits an infinite chain, it admits an infinite chain only using $\mathcal{R}_{\mathsf{TFO}}$-rules or only using $\mathcal{R}_{\mathsf{PHO}}$-rules for the topmost steps.*

Using this result, sufficient conditions to split off $\mathcal{R}_{\mathsf{TFO}}$ entirely are explored, see [5] for details.

5 Conclusion

Summary We have summarised the two main approaches for higher-order dependency pairs for the two higher-order formalisms commonly considered when studying higher-order termination. The static approach was discussed for Higher-Order Rewrite Systems and the dynamic approach was presented for Algebraic Functional Systems. Differences in the higher-order formalisms and in the dependency pair approaches have been discussed and we have sketched how dependency pairs can be used to show termination of higher-order systems.

Related Work The other main termination technique for higher-order rewriting, which has been automated, is the Higher-Order Recursive Path Ordering. Originally defined by Jouannaud and Rubio [10] it was continuously refined in a long line of research, and thus many variations exist, e.g. [11, 3].

Recently van de Pols semantic approach was also reconsidered in the form of polynomial interpretations. Bofill et al. [4] define the (Higher-Order) Recursive Path and Polynomial Ordering, a (HO)RPO variant which allows to compare some of the function symbols by polynomial interpretations. Baillot and Dal Lago [2] consider higher-order interpretations for Yamadas Simply Typed Term Rewriting Systems [24], to analyse complexity. The main difference in the three implementations of the algebra approach is the treatment of the application symbol. Bofill et al. compare applications with the RPO-part of their ordering and thus do not need to interpret it. Kop and van Raamsdonk interpret application as max. Baillot and Dal Lago interpret application of two terms as the application of their interpretations and thus might require β-reduction to compute the interpretation of a term.

Future Work It is our goal to implement the described techniques, automating them in a termination tool for higher-order rewriting. Moreover, right now the static and the dynamic approach coexist, each with advantages and disadvantages. It is therefore desirable to combine the two. Finally it is noteworthy that none of the discussed methods, even the semantic approach, fully characterises termination. This is somewhat intriguing and might be an interesting point for further investigation.

References

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

[2] P. Baillot and U. Dal Lago. Higher-order interpretations and program complexity. In *Proceedings of the 12th International Workshop on Termination, WST '12*, pages 20–24, Obergurgl, Austria, Feburary 2012.

[3] F. Blanqui, J.-P. Jouannaud, and A. Rubio. The computability path ordering: The end of a quest. In *Proceedings of the 22nd International Workshop on Computer Science Logic, CSL '08*, volume 5213 of *Lecture Notes in Computer Science*, pages 1–14, Bertinoro, Italy, 2008. Springer-Verlag.

[4] M. Bofill, C. Borralleras, E. Rodríguez-Carbonell, and A. Rubio. The recursive path and polynomial ordering. In *Proceedings of the 12th International Workshop on Termination, WST '12*, pages 29–33, Obergurgl, Austria, Feburary 2012.

[5] C. Fuhs and C. Kop. Harnessing first order termination provers using higher order dependency pairs. In *Proceedings of the 8th International Symposium on Frontiers of Combining Systems, FroCoS '11*, volume 6989 of *Lecture Notes in Artificial Intelligence*, pages 147–162, Saarbrücken, Germany, 2011. Springer-Verlag.

[6] J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR '04*, volume 3452 of *Lecture Notes in Artificial Intelligence*, pages 301–331, Montevideo, Uruguay, March 2004. Springer-Verlag.

[7] N. Hirokawa and A. Middeldorp. Dependency pairs revisited. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications, RTA '04*, volume 3091 of *Lecture Notes in Computer Science*, pages 249–268, Aachen, Germany, 2004. Springer-Verlag.

[8] N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1,2):172–199, 2005.

[9] J.-P. Jouannaud and M. Okada. A computation model for executable higher-order algebraic specification languages. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science, LICS '91*, pages 350–361, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[10] J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science, LICS '99*, pages 402–411, Trento, Italy, July 1999. IEEE Computer Society Press.

[11] J.-P. Jouannaud and A. Rubio. Higher-order orderings for normal rewriting. In *Proceedings of the 17th International Conference on Rewriting Techniques and Applications, RTA '06*, volume 4098 of *Lecture Notes in Computer Science*, pages 387–399, Seattle, WA, USA, August 2006. Springer-Verlag.

[12] Z. Khasidashvili. Expression reduction systems. In *Proceedings of I. Vekua Institute of Applied Mathematics*, volume 36, pages 200–220, Tblisi, 1990.

[13] J. W. Klop. *Combinatory Reduction Systems*. PhD thesis, University of Utrecht, June 1980.

[14] C. Kop. Simplifying algebraic functional systems. In *Proceedings of the 4th International Conference on Algebraic Informatics, CAI '11*, volume 6742 of *Lecture Notes in Computer Science*, pages 201–215, Linz, Austria, June 2011. Springer-Verlag.

[15] C. Kop and F. van Raamsdonk. Higher order dependency pairs for algebraic functional systems. In *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA '11*, volume 10 of *Leibniz International Proceedings in Informatics*, pages 203–218, Novi Sad, Serbia, 2011. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[16] 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-D(10):2007–2015, 2009.

[17] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, July 1991.

[18] T. Nipkow. Higher-order critical pairs. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science, LICS '91*, pages 342–349, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[19] 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.

[20] 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.

[21] 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, September 2011.

[22] J. C. van de Pol. *Termination of Higher-order Rewrite Systems*. PhD thesis, University of Utrecht, December 1996.

[23] D. A. Wolfram. Rewriting, and equational unification: the higher-order cases. In *Proceedings of the 4th International Conference on Rewriting Techniques and Applications, RTA '91*, volume 488 of *Lecture Notes in Computer Science*, pages 25–36, Como, Italy, April 1991. Springer-Verlag.

[24] T. Yamada. Confluence and termination of simply typed term rewriting systems. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications, RTA '01*, volume 2051 of *Lecture Notes in Computer Science*, pages 338–352, Utrecht, The Netherlands, May 2001. Springer-Verlag.