The Higher-Order Dependency Pair Framework*

Cynthia Kop
University of Innsbruck, Institute of Computer Science, 6020 Innsbruck, Austria
cynthia.kop@uibk.ac.at

In recent years, two different dependency pair approaches have been introduced: the *dynamic* and *static* styles. The static style is based on a computability argument, and is limited to *plain function-passing* systems. The dynamic style has no limitations, but standard techniques to simplify sets of dependency pairs – such as the subterm criterion, usable rules and reduction pairs – are either not applicable or significantly weaker. On the other hand, we can significantly improve the dynamic approach for *local* systems. In this paper, I will discuss how to combine the dynamic and static styles in a single dependency pair *framework*, extending various notions from the first-order setting.

1 Introduction

In modern termination tools for (first-order) term rewriting, the *dependency pair framework* [2, 3] plays a crucial role. In this framework, a set of *dependency pair problems* is gradually simplified by *dependency pair processors* until only trivial problems remain, in which case termination is proved.

In the higher-order setting, there are two main approaches to using dependency pairs, *dynamic* [8, 10] and *static* [9, 11]. Both approaches have different strengths and weaknesses. The *dynamic* approach is always applicable, but gives *collapsing* dependency pairs, which makes standard techniques like the subterm criterion or usable rules hard to apply. On the other hand, the *static* approach is limited to *plain function-passing* systems and surrenders completeness, but does not create collapsing dependency pairs and therefore remains closer to the first-order dependency pair approach.

This paper endeavours to define a higher-order dependency pair *framework*. To avoid double work, this framework is designed to handle both static and dynamic dependency pairs.

Note: the results in this paper have been published in the author's PhD thesis [5, Ch. 7]. The results are also closely related to, but a strict generalisation of, the definitions in [8].

2 Preliminaries

A basic understanding of first-order term rewriting, simple types and λ-calculus is assumed. I will introduce *algebraic functional systems*, but (unlike usual definitions) use explicit meta-variables for matching.

A *type declaration* has the form $[\sigma_1 \times \ldots \times \sigma_n] \to \tau$ with $\sigma_1, \ldots, \sigma_n, \tau$ simple types (τ need not be a base type). Given a set F of *function symbols*, each equipped with a type declaration and sets M, V of *meta-variables* and *variables*, each with a type, *meta-terms* are expressions s such that $s : \sigma$ can be derived for some type σ using the clauses below, and *terms* are meta-terms without meta-variables:

$$\begin{array}{lll}
x : \sigma & \text{if} & x : \sigma \in M \cup V \\
f(s_1, \ldots, s_n) : \tau & \text{if} & f : [\sigma_1 \times \ldots \times \sigma_n] \to \tau \in F \text{ and } s_1 : \sigma_1, \ldots, s_n : \sigma_n \\
\lambda x.s : \sigma \to \tau & \text{if} & x : \sigma \in V \text{ and } s : \tau \\
s \cdot t : \tau & \text{if} & s : \sigma \to \tau \text{ and } t : \sigma
\end{array}$$

*The research described in this paper is supported by the Austrian Science Fund (FWF) international project I963.

Submitted to:
HOR 2014

We consider meta-terms modulo α-equality as usual, and denote $FV(s)$ for the set of free variables of s and $FMV(s)$ for its meta-variables. The *arity* of $f : [\sigma_1 \times \ldots \times \sigma_n] \to \tau \in F$ is n; we will often avoid explicit function notation, and just denote $f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m$ as $f \cdot s_1 \cdots s_m$. A meta-term s is *closed* if $FV(s) = \emptyset$, *linear* if no meta-variable occurs more than once, a *pattern* if meta-variables do not occur at the head of an application and *fully extended* if meta-variables do not occur below an abstraction.[1]

A *substitution* γ maps variables and meta-variables to terms of the same type, and is applied on a term s, notation $s\gamma$, by replacing all x in its domain by $\gamma(x)$ (renaming binders if necessary to avoid capture). Let $s \trianglerighteq t$ if t is a (not necessarily strict) *subterm* of s; subterms may free previously bound variables.

A *rule* is a pair $\ell \Rightarrow r$ of a closed pattern ℓ and a closed meta-term r of the same type, such that ℓ is not a meta-variable, $FMV(r) \subseteq FV(\ell)$, and r is β-normal. The root symbols f of rules $f \cdot \vec{l} \Rightarrow r \in R$ are called *defined symbols*. Given a set of rules R, the relation $\Rightarrow_R$ is the smallest monotonic relation on *terms*[2] which includes the β-reduction relation $\Rightarrow_\beta$ and has $\ell\gamma \Rightarrow_R r\gamma$ for all $\ell \Rightarrow r \in R$ and substitutions γ on domain $FMV(\ell)$. An *algebraic functional system* (*AFS*) is the abstract rewriting system given by the set of terms over F, V, M and the relation $\Rightarrow_R$, and is usually given as the pair (F, R).

Example 1. We consider the AFS twice, with function symbols $\mathsf{o} : \mathsf{nat}$, $\mathsf{s} : [\mathsf{nat}] \to \mathsf{nat}$, $\mathsf{I} : [\mathsf{nat}] \to \mathsf{nat}$, and $\mathsf{twice} : [\mathsf{nat} \to \mathsf{nat} \times \mathsf{nat}] \to \mathsf{nat}$. There are three rewrite rules (with meta-variables G, n):

$$\mathsf{I}(\mathsf{o}) \;\Rightarrow\; \mathsf{o} \qquad \mathsf{I}(\mathsf{s}(n)) \;\Rightarrow\; \mathsf{s}(\mathsf{twice}(\lambda x.\mathsf{I}(x), n)) \qquad \mathsf{twice}(G, n) \;\Rightarrow\; G \cdot (G \cdot n)$$

3 Dependency Pairs and Chains

As mentioned in the introduction, there are different styles of rewriting: the *static* style, which relies on a computability argument, and the *dynamic* style, which gives dependency pairs of a less practical shape, but which is always applicable and gives an equivalence result. However, both are used in the same way; the different styles of dependency pairs just give a different initial set. Let us start with some definitions. We let $F^\sharp$ be the signature F extended with for every function symbol $f : [\vec{\sigma}] \to \tau_1 \to \ldots \tau_m \to \rho$ (with ρ a base type) a fresh symbol $f^\sharp : [\vec{\sigma}] \to \tau_1 \to \ldots \tau_m \to \mathtt{dpsort}$ with $\mathtt{dpsort}$ a fresh base type. For a term s, we define $s^\sharp = f^\sharp(\vec{t})$ if $s = f(\vec{t})$, and $s^\sharp = s$ if s has any other form, including applications $f(\vec{t}) \cdot \vec{q}$.

Definition 1. A *dependency pair* is a pair $\ell \Rrightarrow p$ such that:

- ℓ is a closed pattern of the form $f \cdot l_1 \cdots l_n$ with $f \in F^\sharp$ and all l_i patterns over F (here $m \geq arity(f)$);
- p is a meta-term of the form $B \cdot p_1 \cdots p_m$ with $B \in F^\sharp \cup M$, and all p_j terms over F.

Note that p may have a different type from ℓ, have free variables, and use meta-variables not in ℓ.

Intuitively, the free variables are used to deal with subterms of right-hand sides where bound variables become free; in practice, they must be instantiated with *fresh variables*:

Definition 2. A substitution γ *respects* a dependency pair $\ell \Rrightarrow p$ if its domain consists of all meta-variables in ℓ and p and all free variables in p, and all variables are mapped to distinct fresh variables.

Dependency pairs (and respectful substitutions) are used in the notion of a *chain*:

Definition 3. For a set of dependency pairs P and a set of rules R, an infinite (P,R)-chain is a sequence $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ with each $\rho_i \in P \cup \{\mathsf{beta}\}$ and s_i, t_i terms, and moreover:

1. if $\rho_i = \ell \Rrightarrow p \in P$ then there exists a substitution γ which respects ρ_i, such that $s_i = \ell\gamma$ and $t_i = p\gamma$.

[1]In [5], meta-variables are also permitted to take arguments, which allows us to encode a broad range of higher-order term rewriting formalisms. This possibility was omitted here for simplicity, although similar results hold also for the extension.

[2]*Meta-terms* are just an auxiliary construct to describe rules and, later, dependency pairs, so do not need to be reduced.

2. if $\rho_i = \texttt{beta}$ then $s_i = (\lambda x.q) \cdot u \cdot v_1 \cdots v_k$ and either
   (a) $k > 0$ and $t_i = q[x := u] \cdot v_1 \cdots v_k$, or
   (b) $k = 0$ and there exists a non-variable term v with $q \trianglerighteq v$ and $x \in FV(v)$ and $t_i = v^\sharp[x := u]$;
3. $t_i \Rightarrow_{in}^* s_{i+1}$, that is: if $t_i = f \cdot q_1 \cdots q_n$ then $s_{i+1} = f \cdot u_1 \cdots u_n$ with each $q_j \Rightarrow_R^* u_j$, otherwise $t_i = s_{i+1}$.

At first glance, this seems a bit more complicated than the corresponding notion in the first-order setting. The reason are *collapsing dependency pairs*, where the right-hand side is headed by a meta-variable. These lead to beta-reductions at the root, which may need to be followed by *subterm steps*.

In the first-order setting, particular interest goes to *innermost* chains: chains where subterms are always immediately normalised using $\Rightarrow_R^*$. This might be interesting in the higher-order setting as well, but in this paper, let us instead consider a strategy that is almost the opposite of innermost rewriting.

Definition 4. For ℓ a fixed meta-term, s a term and γ a substitution whose domain contains only meta-variables and variables not in $FV(\ell)$, we say that $s \Rightarrow_R^* \ell\gamma$ by a *formative ℓ-reduction* if $\lambda\vec{x}.\ell$ is not a fully extended linear pattern (where $\{\vec{x}\} = FV(\ell)$), or one of the following clauses holds:

1. $s = \ell\gamma$ and ℓ is a meta-variable;
2. $s = a \cdot s_1 \cdots s_n$ and $\ell = a \cdot l_1 \cdots l_n$ and each $s_i \Rightarrow_R^* l_i\gamma$ by a formative l_i-reduction for $a \in F^\sharp \cup V$;
3. $s = \lambda x.s'$ and $\ell = \lambda x.l'$ and $s' \Rightarrow_R^* l'\gamma$ by a formative l'-reduction (with x of course not used in γ);
4. $s = (\lambda x.t) \cdot q \cdot \vec{u}$, and $t[x := q] \cdot \vec{u} \Rightarrow_R^* \ell\gamma$ by a formative ℓ-reduction;
5. ℓ is not a meta-variable and there are $\ell' \Rightarrow r' \in R$ and δ such that $s \Rightarrow_R^* \ell'\delta$ by a formative ℓ'-reduction and $r\delta \Rightarrow_R^* \ell\gamma$ by a formative ℓ-reduction which does not use clause 5.

The key point of this definition is 1: when reducing to a meta-variable X, we are not allowed to take any intermediate steps other than β-reductions, which must be done immediately. Essentially, we take only those steps which are needed to create a pattern of the form $\ell\gamma$ for some γ.

We say a (P,R)-chain $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ is *formative* if for all i: if $\rho_{i+1} = \ell \Rightarrow p$ then $t_i \Rightarrow_R^* s_{i+1}$ by a formative ℓ-reduction. The chain is *minimal* if the strict subterms of all t_i are terminating in $\Rightarrow_R$.

3.1 Static Dependency Pairs

For an AFS with *base output types* (that is, for all $f : [\sigma_1 \times \ldots \times \sigma_n] \to \tau \in F$ the type τ is base) which is *plain function-passing* (that is, if $f \cdot \vec{l} \Rightarrow r \in R$ and $X \in FMV(r)$ then either X has base type, or X is one of the l_i), the set of *static dependency pairs* $\mathtt{SDP}(R)$ is defined as the set of all dependency pairs $f^\sharp(\vec{l}) \Rightarrow g^\sharp(p'_1, \ldots p'_n)$ where $f(\vec{l}) \Rightarrow r \in R$ and $r \trianglerighteq g(\vec{p})$ and g is the root symbol of some rule in R and each p'_i is p_i with free variables replaced by a corresponding meta-variable.

Example 2. The static dependency pairs for the system from Example 1 are:

$$\mathsf{I}^\sharp(\mathsf{s}(n)) \;\Rightarrow\; \mathsf{twice}^\sharp(\lambda x.\mathsf{I}(x), n) \qquad \mathsf{I}^\sharp(\mathsf{s}(n)) \;\Rightarrow\; \mathsf{I}^\sharp(m)$$

Theorem 5. [9, 11, 5] A plain function-passing AFS with base output types and rules R is terminating if there is no infinite minimal formative $(\mathtt{SDP}(R), R)$-chain.

This is not an *if and only if* because the right-hand sides may introduce fresh meta-variables (like m in Example 2). If no fresh meta-variables are introduced, then we do have an equivalence.

3.2 Dynamic Dependency Pairs

For an AFS R, let $R^{\mathtt{sat}}$ (the *β-saturated rules*) be R extended with, for every rule $\ell \Rightarrow \lambda x_1 \ldots x_n.r \in R$ with $n \geq 0$ and r not an abstraction, the n rules $\ell \cdot Z_1 \Rightarrow \lambda x_2 \ldots x_n.r[x_1 := Z_1]$, $\ldots$, $\ell \cdot Z_1 \cdots Z_n \Rightarrow r[x_1 := Z_1, \ldots, Z_n]$, where the Z_i are fresh meta-variables. Let $R^{\mathtt{full}}$ contain all elements of $R^{\mathtt{sat}}$ and additionally for all rules $\ell \Rightarrow r \in R^{\mathtt{sat}}$ of composed type, but where r is not an abstraction, all well-typed rules $\ell \cdot Z_1 \cdots Z_n \Rightarrow r \cdot Z_1 \cdots Z_n$. The set of *dynamic dependency pairs* $\mathtt{DDP}(R)$ contains all dependency pairs $f^\sharp \cdot \vec{l} \Rightarrow p^\sharp$ where $f \cdot \vec{l} \Rightarrow r \in R^{\mathtt{sat}}$ and $r \trianglerighteq p = a \cdot \vec{p}$ with a either a defined symbol or a meta-variable.

Example 3. For R from Example 1, $R^{\mathtt{full}} = R$, as all symbols have base output type. $\mathtt{DDP}(R)$ contains:

$$\begin{array}{rclrcl} \mathsf{I}^\sharp(\mathsf{s}(n)) & \Rightarrow & \mathsf{twice}^\sharp(\lambda x.\mathsf{I}(x), n) & \mathsf{twice}^\sharp(G, n) & \Rightarrow & G \cdot (G \cdot n) \\ \mathsf{I}^\sharp(\mathsf{s}(n)) & \Rightarrow & \mathsf{I}^\sharp(x) & \mathsf{twice}^\sharp(G, n) & \Rightarrow & G \cdot n \end{array}$$

Here, x is a *variable*, not a meta-variable, so can only be instantiated by fresh variables in a chain.

Theorem 6. [5] An AFS R is terminating iff there is no infinite minimal formative $(\mathtt{DDP}(R), R^{\mathtt{full}})$-chain.

4 The Higher-order Dependency Pair Framework (for Termination)

Now, given an AFS (F, R), we *assume given* initial sets $\mathtt{DP}$ and R' with the following property:

If there is no infinite minimal formative $(\mathtt{DP}, R')$*-chain, then* (F, R) *is terminating.*

These initial sets may be given by either the dynamic or the static approach.

Definition 7. A *dependency pair problem* (DP problem) is a tuple (P, R, f_1, f_2) where P is a set of dependency pairs, R a set of rules, $f_1 \in \{\mathtt{m}, \mathtt{a}\}$ (minimal, arbitrary) and $f_2 \in \{\mathtt{f}, \mathtt{a}\}$ (formative, arbitrary). A DP problem (P, R, f_1, f_2) is called *finite* if there is no infinite (P, R)-chain which moreover is minimal if $f_1 = \mathtt{m}$ and formative if $f_2 = \mathtt{f}$. A *dependency pair processor* is a function which takes a DP problem as input and returns a (possibly empty) set of DP problems. A processor *proc* is *sound* if, for all DP problems A: if all $B \in proc(A)$ are finite, then A is finite.

The *dependency pair framework*, now, is the following non-deterministic algorithm:

1. start with $A := \{(\mathtt{DP}, R')\}$;
2. select a DP problem $X \in A$ and choose a sound processor *proc*;
3. update $A := (A \setminus \{X\}) \cup proc(X)$
4. if $A = \emptyset$ then conclude termination, otherwise continue with 2.

(We could also use the framework to prove *non-termination*, but this is omitted for space reasons.)

There are various processors; to name some from [5]: the *dependency graph*, the *subterm criterion*, *first-order splitting* and *formative rules*. The last of these removes elements from the set R; the others map to DP problems with smaller P. Similar to the first-order setting, several processors are based on *reduction triples*. However, here we must take care: due to subterm steps, we have some extra requirements.

5 Reduction Triple Processors

To start, let us define a counterpart to the first-order *reduction pairs*:

Definition 8. A *reduction triple* $(\succsim, \succeq, \succ)$ consists of two *quasi-orderings* $\succsim, \succeq$ and a *well-founded ordering* $\succ$ such that $\succsim$ and $\succeq$ are compatible with $\succ$ (so both $\succsim \cdot \succ \subseteq \succ$ and $\succeq \cdot \succ \subseteq \succ$), $\succsim$ is monotonic and orients $\Rightarrow_\beta$, and $\succsim, \succeq, \succ$ are all *meta-stable*.

Here, a relation R is meta-stable if it is preserved under variable renaming and $\ell\gamma \mathrel{R} r\gamma$ whenever $\ell \mathrel{R} r$ and ℓ is a pattern of the form $f \cdot \vec{l}$ with $f \in F$ and γ is a substitution on domain $FMV(\ell) \cup FMV(r)$.

We don't directly apply reduction triples on the sets in a DP problem, but rather on *ordering problems*:

Definition 9. The ordering problem for (P_1, P_2, R, f) with $f \in \{\mathtt{f}, \mathtt{a}\}$ is:

- $(P_1, P_2, R, \mathsf{none})$ if all dependency pairs in $P_1 \cup P_2$ are non-collapsing;

- $(P_1, P_2, R, \mathsf{subterm})$ if some dependency pair in $P_1 \cup P_2$ is collapsing, and $f = \mathtt{a}$ or some element of $P_1 \cup P_2 \cup R$ is not *local*, where *local* means: both left-linear and fully extended;
- $(\{\ell \Rrightarrow \mathsf{tag}(p) \mid \ell \Rrightarrow p \in P_1\}, \{\ell \Rrightarrow \mathsf{tag}(p) \mid \ell \Rrightarrow p \in P_2\}, \{\ell \Rightarrow \mathsf{tag}(r) \mid \ell \Rightarrow r \in R\} \cup R_{\mathtt{untag}}, \mathsf{tagsub})$ if any dependency pair in $P_1 \cup P_2$ is collapsing, and $f = \mathtt{f}$ and all elements are $P_1 \cup P_2 \cup R$ are local. Here, F^- is a new signature which contains a symbol $f^- : \sigma$ for all $f : \sigma \in F$ such that f occurs between a bound variable and its binder in a right-hand side of P or R (so $r \trianglerighteq q = f(\vec{q})$ and $FV(q) \not\subseteq FV(r)$ for $(\ell, r) \in P \cup R$). Also, $\mathsf{tag}(s)$ replaces all occurrences of symbols f between a bound variable and its binder by f^-. The set $R_{\mathtt{untag}}$ consists of rules $f^-(x_1, \dots, x_n) \Rightarrow f(x_1, \dots, x_n)$ for $f^- : [\sigma_1 \times \dots \times \sigma_n] \to \tau \in F^-$ and all $x_i \in M$.

A reduction triple $(\succsim, \succeq, \succ)$ orients an ordering problem $(A_1, A_2, B, prop)$ if:

- $\ell \succ p$ for all $\ell \Rrightarrow p \in A_1$ and $\ell \succeq p$ for all $\ell \Rrightarrow p \in A_2$ and $\ell \succsim r$ for all $\ell \Rightarrow r \in B$;
- if $prop = \mathsf{subterm}$, then $\succsim$ satisfies the *subterm property*: for all s, t: if $s \trianglerighteq t$ then there is a substitution γ with domain $FV(t) \setminus FV(s)$ such that $s \succsim t^\sharp \gamma$;
- if $prop = \mathsf{tagsub}$, then $\succsim$ satisfies the *tagged subterm property*: for all $x \in V$ and terms s, t, q with $s \trianglerighteq q \neq x$ and $x \in FV(q)$, there is a substitution γ with $\mathsf{tag}((\lambda x.s) \cdot t) \succsim \mathsf{tag}(q^\sharp[x := t]\gamma)$.

Note that requiring the subterm property is painful, for it makes it impossible to use *argument filterings*, where some arguments of a function symbol are not regarded by reduction triples. The *tagged* subterm property, however, is a lot weaker, as it only affects symbols between a variable and its binder.

Theorem 10. Fixing $(\succsim, \succeq, \succ)$, a processor which maps (P, R, f_1, f_2) to the following result, is sound:

- $\{(P_2, R, f_1, f_2)\}$ if $P = P_1 \uplus P_2$ and $(\succsim, \succeq, \succ)$ orients the ordering problem for (P_1, P_2, R, f_2);
- $\{(P, R, f_1, f_2)\}$ otherwise.

The reason to define *ordering problems*, rather than directly having different cases in the definition of the processor, is because ordering problems can be reused in other processors based on reduction triples. For an example, let us consider *usable and formative rules with respect to an argument filtering*.

Definition 11. An *argument filtering* is a function π which maps each symbol $f : [\sigma_1 \times \dots \times \sigma_n] \to \tau$ to a subset of $\{1, \dots, n\}$. Given s, R and π, let $UR(s, R, \pi)$ be the smallest subset of R such that:

- $UR(s, R, \pi) = R$ if R is not finitely branching (that is, if some s has infinitely many direct reducts);
- if $\ell \Rightarrow r \in UR(s, R, \pi)$ then $UR(r, R, \pi) \subseteq UR(s, R, \pi)$;
- if $s = f \cdot s_1 \cdots s_m$ with $arity(f) = n \leq m$, then $UR(s_i, R, \pi) \subseteq UR(s, R, \pi)$ for all $i \in \pi(f) \cup \{n+1, \dots, m\}$, and $UR(s, R, \pi)$ also contains all rules of the form $f \cdot l_1 \cdots l_m \Rightarrow r \in R$ (same m);
- if $s = x \cdot s_1 \cdots s_n$ with x a variable, then $UR(s_i, R, \pi) \subseteq UR(s, R, \pi)$ for all i;
- if $s = \lambda x.t$, then $UR(t, R, \pi) \subseteq UR(s, R, \pi)$;
- if $s = X \cdot s_1 \cdots s_n$ and $n > 0$, then $UR(s, R, \pi) = R$ for $X \in M$ (if $n = 0$, then $UR(X, R, \pi) = \emptyset$).

Similarly, for a *pattern* s, let $FR(s, R, \pi)$ be the smallest subset of R such that:

- $FR(s, R, \pi) = R$ if s is not linear or not fully extended;
- if $\ell \Rightarrow r \in FR(s, R, \pi)$ then $FR(\ell, R, \pi) \subseteq FR(s, R, \pi)$;
- if $s = f \cdot s_1 \cdots s_m$ with $arity(f) = n \leq m$, then $FR(s_i, R, \pi) \subseteq FR(s, R, \pi)$ for all $i \in \pi(f) \cup \{n+1, \dots, m\}$ and $FR(s, R, \pi)$ contains all rules of the form $\ell \Rightarrow f \cdot r_1 \cdots r_m$ (same m);
- if $s = x \cdot s_1 \cdots s_n$ with x a variable, then $FR(s_i, R, \pi) \subseteq FR(s, R, \pi)$ for all i;
- if $s = \lambda x.t$, then $FR(t, R, \pi) \subseteq FR(s, R, \pi)$;
- if $s : \sigma$ and $s \notin M$, then $FR(s_i, R, \pi)$ contains all rules $\ell \Rightarrow X \cdot \vec{r} \in R$ where $\ell : \sigma$ and $X \in M$.

We define $UR(P, R, \pi) = \bigcup_{\ell \Rrightarrow p \in P} UR(p, R, \pi)$ and $FR(P, R, \pi) = \bigcup_{\ell \Rrightarrow p \in P} FR(\ell, R, \pi)$.

Claim 12. Let π be an argument filtering, and $(\succsim, \succeq, \succ)$ a reduction triple such that for all s_i, t_i we have: $f(\dots, s_i, \dots) \succsim f(\dots, t_i, \dots)$ if $i \notin \pi(f)$. A processor which maps (P, R, f_1, f_2) to $\{(P_2, R, f_1, f_2)\}$ if $P = P_1 \uplus P_2$, $f_1 = \mathtt{m}$, $f_2 = \mathtt{f}$ and $(\succsim, \succeq, \succ)$ orients the ordering problem for $(P_1, P_2, UR(P, FR(P, R, \pi), \pi), f_2)$, and to $\{(P, R, f_1, f_2)\}$ otherwise, is sound.

(The proof of this claim is still work in progress, but it seems *true.)*

6 Conclusions

In this paper, we have seen a higher-order version of the dependency pair framework, as it is typically used in first-order termination analysis. Although we still have to choose, at the beginning, whether to use static or dynamic dependency pairs, the framework itself is the same for either choice.

Compared to the dependency pair *approach* in [8], the framework has significant advantages. In particular, in [8], tags are carried along in DP problems; here, thanks to the *formative* flag, they are restricted to processors with reduction triples, which simplifies reasoning for other processors. Even for processors based on reduction pairs (like the reduction pair processor with usable rules), we do not have to define special cases for the various subterm properties. The formative flag also makes it possible to use formative rules with respect to an argument filtering, which was not possible with the definitions of [8].

This version of the DP framework is implemented in the higher-order termination tool Wanda [4]. This tool uses a more general formalism, AFSMs rather than AFSs, where meta-variables can take arguments (like in [5]) and right-hand sides of rules do not need to be β-normal; however, she is optimised for AFSs as described here. Additionally, following [6], the static approach is extended: base output types are not needed if we use $R^{\mathtt{full}}$ like in the dynamic approach, and "plain function passing" can be weakened to allow functional meta-variables at *accessible* positions of the left-hand sides.

References

[1] C. Fuhs & C. Kop (2014): *First-Order Formative Rules*. In: *Proceedings of RTA-TLCA '14*. To appear.

[2] J. Giesl, R. Thiemann, P. Schneider-Kamp & S. Falke (2006): *Mechanizing and Improving Dependency Pairs*. *Journal of Automated Reasoning* 37(3).

[3] N. Hirokawa & A. Middeldorp (2005): *Automating the dependency pair method*. *Information and Computation* 199(1-2).

[4] C. Kop: *WANDA – a higher order termination tool*. `http://www.few.vu.nl/~kop/code.html`.

[5] C. Kop (2012): *Higher Order Termination*. Ph.D. thesis, Vrije Universiteit Amsterdam.

[6] C. Kop (2013): *Static Dependency Pairs with Accessibility*. `http://cl-informatik.uibk.ac.at/users/kop/static.pdf`.

[7] C. Kop & F. van Raamsdonk (2011): *Higher Order Dependency Pairs for Algebraic Functional Systems*. In M. Schmidt-Schauß, editor: *Proceedings of RTA '11*, *LIPIcs* 10, Dagstuhl.

[8] C. Kop & F. van Raamsdonk (2012): *Dynamic Dependency Pairs for Algebraic Functional Systems*. *Logical Methods in Computer Science* 8(2). Included in the Special Issue for RTA '11.

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

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

[11] S. Suzuki, K. Kusakari & F. Blanqui (2011): *Argument Filterings and Usable Rules in Higher-Order Rewrite Systems*. *IPSJ Transactions on Programming* 4(2).