Improving Static Dependency Pairs for Higher-Order Rewriting

Carsten Fuhs[1] and Cynthia Kop[2]

1 **Birkbeck, University of London, United Kingdom** carsten@dcs.bbk.ac.uk
2 **Radboud University Nijmegen, The Netherlands** c.kop@cs.ru.nl

Abstract

We revisit the static dependency pair method for termination of higher-order term rewriting. In this extended abstract, we propose a static dependency pair framework based on an extended notion of computable dependency chains that harnesses the computability-based reasoning used in the soundness proof of static dependency pairs. This allows us to propose a new termination proving technique to use in combination with static DPs: the *computable* subterm criterion.

Digital Object Identifier 10.4230/LIPIcs.WST.2018.

1 Introduction

This paper deals with higher-order term rewriting with β-reduction and λ-abstractions. Here a particular topic of interest is *termination*, the property that all (well-formed) terms have only finite reductions. In the first-order setting, the *Dependency Pair (DP) framework* [8] has proven to be an extremely successful foundation for automated termination analysis tools. While several DP approaches (static [12, 14] and dynamic [13, 10]) exist for higher-order rewriting, so far a general DP framework has been proposed only in the PhD thesis [9]. We build on ideas from [2, 9] to propose such a DP framework, here specialised to static DPs, and include a completely new processor which can offer a simple syntactic termination criterion.

2 Algebraic Functional Systems with Meta-variables

Henceforth, we shall assume familiarity with term rewriting, simple types and the λ-calculus. We use a simplified version of *Algebraic Functional Systems with Meta-variables (AFSMs)* that Kop [9] proposes to capture a number of higher-order rewrite formalisms (cf. [9, Ch. 3]).

We fix disjoint sets $\mathcal{F}$ of *function symbols* and $\mathcal{V}$ of *variables*, each symbol a equipped with a type σ. We also fix a set $\mathcal{M}$, disjoint from $\mathcal{F}$ and $\mathcal{V}$, of *meta-variables*, each equipped with a *type declaration* $[\sigma_1 \times \cdots \times \sigma_k] \rightarrow \tau$ (where τ and all σ_i are simple types). *Meta-terms* are expressions s where $s : \sigma$ can be derived for some type σ by the following clauses:

(V)	$x : \sigma$	if	$x : \sigma \in \mathcal{V}$	(@)	$s\ t : \tau$	if	$s : \sigma \rightarrow \tau$ and $t : \sigma$
(F)	$\mathtt{f} : \sigma$	if	$\mathtt{f} : \sigma \in \mathcal{F}$	(Λ)	$\lambda x.s : \sigma \rightarrow \tau$	if	$x : \sigma \in \mathcal{V}$ and $s : \tau$

(M) $Z[s_1, \dots, s_k] : \tau$ if $Z : [\sigma_1 \times \cdots \times \sigma_k] \rightarrow \tau \in \mathcal{M}$ and $s_1 : \sigma_1, \dots, s_k : \sigma_k$

Terms are meta-terms without meta-variables, so derived without clause (M). *Patterns* are meta-terms where all meta-variable occurrences have the form $Z[x_1, \dots, x_k]$ with all x_i distinct variables. The λ binds variables as in the λ-calculus. Unbound variables are called *free*, $FV(s)$ is the set of free variables in s, and $FMV(s)$ is the set of meta-variables occurring in s. A meta-term s is *closed* if $FV(s) = \emptyset$. Meta-terms are considered modulo α-conversion. Application (@) is left-associative; abstractions (Λ) extend as far to the right as possible. A meta-term s *has type* σ if $s : \sigma$; it *has base type* if $\sigma \in \mathcal{S}$, the set of *sorts*. A meta-term s has a *sub-meta-term* t (*subterm* if t is a term), written $s \trianglerighteq t$, if (a) $s = t$, (b) $s = \lambda x.s'$ and $s' \trianglerighteq t$, (c) $s = s_1\ s_2$ and $s_1 \trianglerighteq t$ or $s_2 \trianglerighteq t$, or (d) $s = Z[s_1, \dots, s_k]$ and some $s_i \trianglerighteq t$.

A *meta-substitution* is a type-preserving function γ from variables and meta-variables to meta-terms; if $Z : [\sigma_1 \times \cdots \times \sigma_k] \to \tau$ then $\gamma(Z)$ has the form $\lambda y_1 \ldots y_k.u : \sigma_1 \to \ldots \to \sigma_k \to \tau$. Let $\mathtt{dom}(\gamma) = \{x \in \mathcal{V} \mid \gamma(x) \neq x\} \cup \{Z \in \mathcal{M} \mid \gamma(Z) \neq \lambda y_1 \ldots y_k.Z[y_1, \ldots, y_k]\}$ (the *domain* of γ). We let $[b_1 := s_1, \ldots, b_n := s_n]$ be the meta-substitution γ with $\gamma(b_i) = s_i$, $\gamma(z) = z$ for $z \in \mathcal{V} \setminus \{\vec{b}\}$, and $\gamma(Z) = \lambda y_1 \ldots y_k.Z[y_1, \ldots, y_k]$ for $Z \in \mathcal{M} \setminus \{\vec{b}\}$. A *substitution* is a meta-substitution mapping everything in its domain to terms. The result $s\gamma$ of applying a meta-substitution γ to a meta-term s is obtained recursively (with implicit α-conversion):

$$
\begin{array}{llllllllll}
x\gamma & = & \gamma(x) & \text{if} & x \in \mathcal{V} & (s\ t)\gamma & = & (s\gamma)\ (t\gamma) & & \\
\mathtt{f}\gamma & = & \mathtt{f} & \text{if} & \mathtt{f} \in \mathcal{F} & (\lambda x.s)\gamma & = & \lambda x.(s\gamma) & \text{if} & \gamma(x) = x \wedge x \notin FV(s\gamma)
\end{array}
$$

$$Z[s_1, \ldots, s_k]\gamma = t[x_1 := s_1\gamma, \ldots, x_k := s_k\gamma] \text{ if } \gamma(Z) = \lambda x_1 \ldots x_k.t$$

Essentially, applying a meta-substitution with meta-variables in its domain combines a substitution with a β-development, e.g., $X[\mathtt{nil}, \mathtt{0}][X := \lambda x.\mathtt{plus}\ (\mathtt{len}\ x)]$ equals $\mathtt{plus}\ (\mathtt{len}\ \mathtt{nil})\ \mathtt{0}$.

A *rule* is a pair $\ell \Rightarrow r$ of *closed* meta-terms of the same type both in β-normal form with ℓ a pattern of the form $\mathtt{f}\ \ell_1 \cdots \ell_n$ with $\mathtt{f} \in \mathcal{F}$, and $FMV(r) \subseteq FMV(\ell)$. A set of rules $\mathcal{R}$ induces a rewrite relation $\Rightarrow_{\mathcal{R}}$ as the smallest monotonic relation on terms that includes β-reduction (denoted as $\Rightarrow_\beta$) and has $\ell\delta \Rightarrow_{\mathcal{R}} r\delta$ whenever $\ell \Rightarrow r \in \mathcal{R}$ and δ is a substitution on domain $FMV(\ell)$. Rewriting is allowed at any position of a term, even below a λ. $\mathcal{R}$ is terminating if there is no infinite reduction $s_0 \Rightarrow_{\mathcal{R}} s_1 \Rightarrow_{\mathcal{R}} \ldots$. The set $\mathcal{D} \subseteq \mathcal{F}$ of *defined symbols* consists of those $\mathtt{f} \in \mathcal{F}$ such that a rule $\mathtt{f}\ \ell_1 \cdots \ell_n \Rightarrow r$ exists.

An AFSM is a pair $(\mathcal{F}, \mathcal{R})$; types of (meta-)variables can be derived from context.

▶ **Example 1** (Ordinal recursion). Let $\mathcal{F}$ contain at least $\mathtt{0} : \mathtt{ord}$, $\mathtt{s} : \mathtt{ord} \to \mathtt{ord}$, $\mathtt{lim} : (\mathtt{nat} \to \mathtt{ord}) \to \mathtt{ord}$ for ordinals, $\mathtt{zero} : \mathtt{nat}$, $\mathtt{succ} : \mathtt{nat} \to \mathtt{nat}$ for $\mathbb{N}$, and the symbol $\mathtt{rec} : \mathtt{ord} \to \mathtt{nat} \to (\mathtt{ord} \to \mathtt{nat} \to \mathtt{nat}) \to ((\mathtt{nat} \to \mathtt{ord}) \to (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}) \to \mathtt{nat}\}$. Let $\mathcal{R}$ be:

$$
\begin{array}{lll}
\mathtt{rec}\ \mathtt{0}\ K\ F\ G & \Rightarrow & K, \qquad \mathtt{rec}\ (\mathtt{s}\ X)\ K\ F\ G \Rightarrow F\ X\ (\mathtt{rec}\ X\ K\ F\ G), \\
\mathtt{rec}\ (\mathtt{lim}\ H)\ K\ F\ G & \Rightarrow & G\ H\ (\lambda m.\mathtt{rec}\ (H\ m)\ K\ F\ G)
\end{array}
$$

Then $\mathtt{rec}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{zero}\ (\lambda vz.z)\ (\lambda xy.\mathtt{zero}) \Rightarrow_{\mathcal{R}} (\lambda vz.z)\ \mathtt{0}\ (\mathtt{rec}\ \mathtt{0}\ \mathtt{zero}\ (\lambda vz.z)\ (\lambda xy.\mathtt{zero})) \Rightarrow_\beta (\lambda z.z)\ (\mathtt{rec}\ \mathtt{0}\ \mathtt{zero}\ (\lambda vz.z)\ (\lambda xy.\mathtt{zero})) \Rightarrow_\beta \mathtt{rec}\ \mathtt{0}\ \mathtt{zero}\ (\lambda vz.z)\ (\lambda xy.\mathtt{zero}) \Rightarrow_{\mathcal{R}} \mathtt{zero}$.

3 Computability

A common technique in higher-order termination is Tait and Girard's *computability* notion [15]. There are several ways to define computability predicates; here we follow, e.g., [1, 3, 4, 5] in considering *accessible meta-variables* using a form of the *computability closure* [3]:

▶ **Definition 2** (Accessible arguments). We fix a quasi-ordering $\succeq^{\mathcal{S}}$ on the set of sorts (base types) $\mathcal{S}$ with well-founded strict part $\succ^{\mathcal{S}} := \succeq^{\mathcal{S}} \setminus \preceq^{\mathcal{S}}$. For $\sigma \equiv \sigma_1 \to \ldots \to \sigma_m \to \kappa$ (with $\kappa \in \mathcal{S}$) and sort ι, let $\iota \succeq^{\mathcal{S}}_{+} \sigma$ if $\iota \succeq^{\mathcal{S}} \kappa$ and each $\iota \succ^{\mathcal{S}}_{-} \sigma_i$, and let $\iota \succ^{\mathcal{S}}_{-} \sigma$ if $\iota \succ^{\mathcal{S}} \kappa$ and each $\iota \succeq^{\mathcal{S}}_{+} \sigma_i$. (The relation $\iota \succeq^{\mathcal{S}}_{+} \sigma$ corresponds to "ι occurs only positively in σ" in [1, 4, 5].)

For $\mathtt{f} : \sigma_1 \to \ldots \to \sigma_m \to \iota \in \mathcal{F}$, let $Acc(\mathtt{f}) = \{i \mid 1 \leq i \leq m \wedge \iota \succeq^{\mathcal{S}}_{+} \sigma_i\}$. For $x : \sigma_1 \to \ldots \to \sigma_m \to \iota \in \mathcal{V}$, let $Acc(x) = \{i \mid 1 \leq i \leq m \wedge \sigma_i$ has the form $\tau_1 \to \ldots \to \tau_n \to \kappa$ for some $n \in \mathbb{N}$ with $\iota \succeq^{\mathcal{S}} \kappa\}$. We write $s \trianglerighteq_{\mathtt{acc}} t$ if either $s = t$, or $s = \lambda x.s'$ and $s' \trianglerighteq_{\mathtt{acc}} t$, or $s = a\ s_1 \cdots s_n$ with $a \in \mathcal{F} \cup \mathcal{V}$ and $s_i \trianglerighteq_{\mathtt{acc}} t$ for some $i \in Acc(a)$.

▶ **Theorem 3** ($\mathcal{R}$-computability). *For $\mathcal{R}$ a set of rules, there exists a predicate "$\mathcal{R}$-computable" on terms which satisfies the following properties:*

- *$s : \sigma \to \tau$ is $\mathcal{R}$-computable iff $s\ t$ is $\mathcal{R}$-computable whenever $t : \sigma$ is $\mathcal{R}$-computable;*
- *$s : \iota$ for ι a sort is $\mathcal{R}$-computable iff (1) s is terminating under $\Rightarrow_{\mathcal{R}} \cup \Rrightarrow_I$ and (2) if $s \Rightarrow^*_{\mathcal{R}} \mathtt{f}\ s_1 \cdots s_m$ then s_i is $\mathcal{R}$-computable for all $i \in Acc(\mathtt{f})$. Here, $\mathtt{f}\ s_1 \cdots s_m \Rrightarrow_I s_i\ t_1 \cdots t_n$ if both sides have (possibly different) base types, $i \in Acc(\mathtt{f})$, and all t_j are $\mathcal{R}$-computable.*

The above notion of computability is adapted from [1, 3, 4, 5] to account for AFSMs. It is an instance of a *strong computability predicate* following [11], identified by a syntactic criterion. This instance gives a more liberal restriction (in our Def. 9) than their default predicate SC, which is directly used to define the "plain function passing" criterion in [12, 14].

▶ **Example 4.** Consider a quasi-ordering $\succeq^{\mathcal{S}}$ such that $\texttt{ord} \succ^{\mathcal{S}} \texttt{nat}$. In Ex. 1, we then have $\texttt{ord} \succeq^{\mathcal{S}}_{+} \texttt{nat} \rightarrow \texttt{ord}$. Therefore, $1 \in \mathit{Acc}(\texttt{lim})$, which gives $\texttt{lim}\ H \trianglerighteq_{\texttt{acc}} H$.

4 Static DPs for Accessible Function Passing AFSMs

We will adapt static DPs to our AFSM formalism and propose an alternative applicability criterion. Similar to DPs in the first-order setting, static DPs employ *marked symbols*:

▶ **Definition 5** (Marked symbols, DPs)**.** Define $\mathcal{F}^\sharp := \mathcal{F} \uplus \{\texttt{f}^\sharp : \sigma \mid \texttt{f} : \sigma \in \mathcal{D}\}$. For a meta-term s, let $s^\sharp := \texttt{f}^\sharp\ s_1 \cdots s_k$ if $s = \texttt{f}\ s_1 \cdots s_k$ with $\texttt{f} \in \mathcal{D}$; let $s^\sharp := s$ otherwise. A *DP* is a pair $\ell \Rrightarrow p$ where ℓ is a closed pattern $\texttt{f}\ \ell_1 \cdots \ell_m$, p is a meta-term $\texttt{g}\ p_1 \cdots p_k$, and both ℓ and p are β-normal and have (possibly different) base types.

The original static approaches define DPs as pairs $\ell^\sharp \Rrightarrow p^\sharp$ with $\ell \Rightarrow r$ a rule and p a subterm $\texttt{g}\ p_1 \cdots p_k$ of r (their rules use *terms*, not *meta-terms*). This can set bound variables from r free in p. Here, we replace such variables by meta-variables. (So our "variables" mimic (λ-)*bound* variables in functional programming, and our "meta-variables" *free* variables.)

▶ **Definition 6** (*SDP*)**.** For a meta-term s, $\mathit{metafy}(s)$ denotes s with all free variables replaced by corresponding fresh meta-variables. For an AFSM $(\mathcal{F}, \mathcal{R})$, $\mathit{SDP}(\mathcal{R}) = \{\ell^\sharp \Rrightarrow \mathit{metafy}(p^\sharp) \mid \ell \Rightarrow r \in \mathcal{R} \wedge r \trianglerighteq p \wedge \ell$ and p have base types $\wedge\ p$ has the form $\texttt{g}\ p_1 \cdots p_k$ for some $\texttt{g} \in \mathcal{D}\}$.

Right-hand sides of static DPs may contain meta-variables that do not occur on the left:

▶ **Example 7.** For Ex. 1, we obtain $\mathit{SDP}(\mathcal{R}) = \{\texttt{rec}^\sharp\ (\texttt{s}\ X)\ K\ F\ G \Rrightarrow \texttt{rec}^\sharp\ X\ K\ F\ G$, $\texttt{rec}^\sharp\ (\texttt{lim}\ H)\ K\ F\ G \Rrightarrow \texttt{rec}^\sharp\ (H\ M)\ K\ F\ G\}$.

Dependency chains capture sequences of function calls, similar to the first-order setting:

▶ **Definition 8** (Dependency chain, minimal chain)**.** Let $\mathcal{P}$ be a set of DPs and $\mathcal{R}$ be a set of rules. A (finite or infinite) $(\mathcal{P}, \mathcal{R})$-*dependency chain* (or just $(\mathcal{P}, \mathcal{R})$-chain) is a sequence $[(\rho_0, s_0, t_0), (\rho_1, s_1, t_1), \ldots]$ where each $\rho_i \in \mathcal{P}$ and all s_i, t_i are terms, such that for all i:

1. if $\rho_i = \ell_i \Rrightarrow p_i$, then there exists a substitution γ on domain $\mathit{FMV}(\ell_i) \cup \mathit{FMV}(p_i)$ such that $s_i = \ell_i\gamma$ and $t_i = p_i\gamma$; and
2. we can write $t_i = \texttt{f}\ u_1 \cdots u_n$ with $\texttt{f} \in \mathcal{F}^\sharp$, $s_{i+1} = \texttt{f}\ w_1 \cdots w_n$ and each $u_j \Rightarrow^*_{\mathcal{R}} w_j$.

A $(\mathcal{P}, \mathcal{R})$-chain is *minimal* if the strict subterms of all t_i are terminating under $\Rightarrow_{\mathcal{R}}$.

Static DPs are *sound* if the AFSM's rules are *accessible function passing (AFP)*. Intuitively: meta-variables of a higher type may occur only in "safe" places in the left-hand sides of rules.

▶ **Definition 9** (Accessible function passing)**.** An AFSM $(\mathcal{F}, \mathcal{R})$ is *accessible function passing (AFP)* if there exists a sort ordering $\succeq^{\mathcal{S}}$ following Def. 2 such that:

- all function symbols $\texttt{f}$ are fully applied in $\mathcal{R}$, i.e., they occur only with the maximum number of arguments permitted by their type;
- for all $\texttt{f}\ \ell_1 \cdots \ell_m \Rightarrow r \in \mathcal{R}$ and all $Z \in \mathit{FMV}(r)$: there are some variables $x_1, \ldots, x_k$ and some i such that $\ell_i \trianglerighteq_{\texttt{acc}} Z[x_1, \ldots, x_k]$.

This definition is strictly more liberal than the notions of *plain function passing* in [12, 14] as adapted to AFSMs; this lets us handle examples like ordinal recursion (Ex. 1) not covered by [12, 14]. However, [12, 14] consider a different formalism, with polymorphism and rules whose left-hand side is not a pattern. Our restriction is closer to the "admissible" rules in [2], which are defined using a pattern computability closure [1]. It is also an instance of the ATRFP notion [11], which is parametrised by a strong computability predicate and accessibility relation.

▶ **Example 10.** The AFSM from Ex. 1 is AFP because of the sort ordering $\mathtt{ord} \succ^{\mathcal{S}} \mathtt{nat}$ (see also Ex. 4), yet it is not plain function passing following [14].

▶ **Theorem 11.** *If $(\mathcal{F}, \mathcal{R})$ is non-terminating and AFP, then there is an infinite minimal $(SDP(\mathcal{R}), \mathcal{R})$-chain.*

This theorem corresponds to results in [2, 11, 12], but imposes a different admissibility restriction: our notion is strictly more liberal than the syntactic criterion in [12], is likely less liberal than the semantic restriction in [11] (although we could not find an example that is ATRFP but not AFP), and mostly (although not entirely) implies the restriction in [2].

The computability inherent in dependency chains using *SDP* lets us strengthen Thm. 11: rather than considering *minimal* chains, we require (some) subterms of all t_i to be *computable*:

▶ **Definition 12.** A $(\mathcal{P}, \mathcal{R})$-chain $[(\ell_0 \Rrightarrow p_0, s_0, t_0), (\ell_1 \Rrightarrow p_1, s_1, t_1), \dots]$ is *$\mathcal{U}$-computable* for a set of rules $\mathcal{U}$ if $\Rightarrow_{\mathcal{U}} \supseteq \Rightarrow_{\mathcal{R}}$, for all i there exists a substitution γ_i with $s_i = \ell_i\gamma_i$ and $t_i = p_i\gamma_i$, and $(\lambda x_1 \dots x_n.v)\gamma_i$ is $\mathcal{U}$-computable for all v such that $p_i \trianglerighteq v$ and $FV(v) = \{x_1, \dots, x_n\}$.

▶ **Theorem 13.** *(a) If an AFSM $(\mathcal{F}, \mathcal{R})$ is non-terminating and AFP, then there is an infinite $\mathcal{R}$-computable $(SDP(\mathcal{R}), \mathcal{R})$-chain. (b) Every $\mathcal{U}$-computable $(\mathcal{P}, \mathcal{R})$-chain is minimal.*

This theorem does not have a true counterpart in the literature. The main result of [11] does require the immediate arguments of each s_i, t_i to be computable, but not other sub-metaterms. Note that the reverse of (a) does *not* hold; terminating AFSMs $\mathcal{R}$ with infinite $\mathcal{R}$-computable $(SDP(\mathcal{R}), \mathcal{R})$-chains do exist [7, Ex. 3.23 (report version 1)].

5 Static DP Framework & Computable Subterm Criterion Processor

The static DP framework follows the first-order DP framework [8], as an extendable framework for proving termination where new termination methods can easily be added as *processors*. In Thm. 16, we will propose a new processor: the *computable subterm criterion*.

Thus far, we have reduced the problem of termination to the non-existence of certain chains. Following the first-order DP framework, we formalise this further via *DP problems*:

▶ **Definition 14** (DP problem). A *DP problem* is a tuple $(\mathcal{P}, \mathcal{R}, m)$ with $\mathcal{P}$ a set of DPs, $\mathcal{R}$ a set of rules, and $m \in \{\mathsf{minimal}, \mathsf{arbitrary}\} \cup \{\mathsf{computable}_{\mathcal{U}} \mid \mathcal{U} \text{ a set of rules}\}$. A DP problem $(\mathcal{P}, \mathcal{R}, m)$ is *finite* if there exists no infinite $(\mathcal{P}, \mathcal{R})$-chain that is $\mathcal{U}$-computable if $m = \mathsf{computable}_{\mathcal{U}}$ or minimal if $m = \mathsf{minimal}$. For the different levels of permissiveness, we use a transitive-reflexive relation $\succeq$ generated by $\mathsf{computable}_{\mathcal{U}} \succeq \mathsf{minimal} \succeq \mathsf{arbitrary}$.

Thm. 13 now becomes: an AFSM $(\mathcal{F}, \mathcal{R})$ is terminating if (but not only if) it is AFP and $(SDP(\mathcal{R}), \mathcal{R}, \mathsf{computable}_{\mathcal{R}})$ is finite. We add a flag value $\mathsf{computable}_{\mathcal{R}}$ over the first-order framework for chains with computability restrictions. The core idea of the DP framework is to simplify a set of DP problems stepwise via *processors* until nothing remains to be proved:

▶ **Definition 15** (Processor). A *dependency pair processor* (or just *processor*) is a function that takes a DP problem and returns a set of DP problems. A processor *Proc* is *sound* if a DP problem M is finite whenever all elements of $Proc(M)$ are finite.

To prove finiteness of a DP problem M: (1) let $A := \{M\}$; (2) while $A \neq \emptyset$: select a $Q \in A$ and a sound processor *Proc*, let $A := (A \setminus \{Q\}) \cup Proc(Q)$. If this terminates, M is a finite DP problem. Many processors are possible; here we present an extension of the subterm criterion [12, 10, 11], dubbed *computable subterm criterion*, that *needs* the new flag.

▶ **Theorem 16** (Computable subterm criterion processor). *Let* $M = (\mathcal{P}_1 \uplus \mathcal{P}_2, \mathcal{R}, \mathsf{computable}_{\mathcal{U}})$ *be a DP problem. A* projection function ν *maps meta-terms to natural numbers such that for all DPs* $\ell \Rrightarrow p \in \mathcal{P}_1 \uplus \mathcal{P}_2$*, the function* $\overline{\nu}$ *with* $\overline{\nu}(\mathtt{f}\ s_1 \cdots s_m) = s_{\nu(\mathtt{f})}$ *is well-defined for* ℓ *and* p*. For meta-terms* s *and* t *of base types, we define* $s \sqsupset t$ *if* $s \neq t$ *and (a)* $s \trianglerighteq_{\mathtt{acc}} t$ *or (b) there exists a meta-variable* Z *with* $s \trianglerighteq_{\mathtt{acc}} Z[x_1, \ldots, x_k]$ *and* $t = Z[t_1, \ldots, t_k]\ s_1 \cdots s_n$*. Then the processor* $\mathit{Proc}_{\mathtt{compsub}}$ *that maps* M *to* $\{(\mathcal{P}_2, \mathcal{R}, \mathsf{computable}_{\mathcal{U}})\}$ *is sound if a projection function* ν *exists with* $\overline{\nu}(\ell) \sqsupset \overline{\nu}(p)$ *for all* $\ell \Rrightarrow p \in \mathcal{P}_1$ *and* $\overline{\nu}(\ell) = \overline{\nu}(p)$ *for all* $\ell \Rrightarrow p \in \mathcal{P}_2$*.*

▶ **Example 17.** $\mathcal{R}$ from Ex. 1 is terminating if $(\mathcal{P}, \mathcal{R}, \mathsf{computable}_{\mathcal{R}})$ with $\mathcal{P} = \mathit{SDP}(\mathcal{R})$ is finite (see Ex. 7). Consider the projection function ν with $\nu(\mathtt{rec}^\sharp) = 1$. As $\mathtt{s}\ X \trianglerighteq_{\mathtt{acc}} X$ and $\mathtt{lim}\ H \trianglerighteq_{\mathtt{acc}} H$, we have $\mathtt{s}\ X \sqsupset X$ and $\mathtt{lim}\ H \sqsupset H\ M$. So $\mathit{Proc}_{\mathtt{compsub}}(\mathcal{P}, \mathcal{R}, \mathsf{computable}_{\mathcal{R}}) = \{(\emptyset, \mathcal{R}, \mathsf{computable}_{\mathcal{R}})\}$. As there are no DPs left, this implies termination of the original $\mathcal{R}$.

6 Conclusion

We have extended the static DP method by a more relaxed applicability criterion and the new *computable subterm criterion*. The full version [7] of the paper has proofs and further extensions, such as *formative* reductions [6, 10], applications to proving non-termination, and dynamic DPs [10] in a unified DP framework with many other processors.

References

1 F. Blanqui. Termination and confluence of higher-order rewrite systems. In *RTA '00*, 2000.
2 F. Blanqui. Higher-order dependency pairs. In *Proc. WST '06*, 2006.
3 F. Blanqui. Termination of rewrite relations on λ-terms based on Girard's notion of reducibility. *Theoretical Computer Science*, 611:50–86, 2016.
4 F. Blanqui, J. Jouannaud, and M. Okada. Inductive-data-type systems. *Theoretical Computer Science*, 272(1-2):41–68, 2002.
5 F. Blanqui, J. Jouannaud, and A. Rubio. The computability path ordering. *Logical Methods in Computer Science*, 11(4), 2015.
6 C. Fuhs and C. Kop. First-order formative rules. In *Proc. RTA-TLCA '14*, 2014.
7 C. Fuhs and C. Kop. The unified higher-order dependency pair framework. Technical Report `arXiv:1805.09390 [cs.LO]`, 2018. `https://arxiv.org/abs/1805.09390`.
8 J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR '04*, 2005.
9 C. Kop. *Higher Order Termination*. PhD thesis, VU Amsterdam, 2012.
10 C. Kop and F. van Raamsdonk. Dynamic dependency pairs for algebraic functional systems. *Logical Methods in Computer Science*, 8(2):10:1–10:51, 2012.
11 K. Kusakari. Static dependency pair method in functional programs. *IEICE Transactions on Information and Systems*, E101.D(6):1491–1502, 2018.
12 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.
13 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.
14 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.
15 W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32(2):187–199, 1967.