Improving Static Dependency Pairs for Higher-Order Rewriting

Carsten Fuhs¹ and Cynthia Kop²

1 Birkbeck, University of London, United Kingdom carsten@cs.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.


1 Introduction

This paper deals with higher-order term rewriting with \( \beta \)-reduction and \( \lambda \)-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 \( \lambda \)-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 \( \sigma \). 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 \( \tau \) and all \( \sigma_i \) are simple types). Meta-terms are expressions \( s \) where \( s : \sigma \) can be derived for some type \( \sigma \) by the following clauses:

\[
\begin{align*}
(V) \quad & x : \sigma \quad \text{if} \quad x : \sigma \in \mathcal{V} \\
(\emptyset) \quad & s \cdot t : \tau \quad \text{if} \quad s : \sigma \rightarrow \tau \quad \text{and} \quad t : \sigma \\
(F) \quad & \xi : \sigma \quad \text{if} \quad \xi : \sigma \in \mathcal{F} \\
(\Lambda) \quad & \lambda x . s : \tau \quad \text{if} \quad x : \sigma \in \mathcal{V} \quad \text{and} \quad s : \tau \\
(M) \quad & Z[s_1, \ldots, s_k] : \tau \quad \text{if} \quad Z : [\sigma_1 \times \cdots \times \sigma_k] \rightarrow \tau \in \mathcal{M} \quad \text{and} \quad s_1 : \sigma_1, \ldots, s_k : \sigma_k
\end{align*}
\]

Terms are meta-terms without meta-variables, so derived without clause (M). Patterns are meta-terms where all meta-variable occurrences have the form \( Z[x_1, \ldots, x_k] \) with all \( x_i \) distinct variables. The \( \lambda \) binds variables as in the \( \lambda \)-calculus. Unbound variables are called free, \( \mathcal{F} \mathcal{V}(s) \) is the set of free variables in \( s \), and \( \mathcal{F} \mathcal{M}(s) \) is the set of meta-variables occurring in \( s \). A meta-term \( s \) is closed if \( \mathcal{F} \mathcal{V}(s) = \emptyset \). Meta-terms are considered modulo \( \alpha \)-conversion.

Application \( (\emptyset) \) is left-associative; abstractions \( (\Lambda) \) extend as far to the right as possible. A meta-term \( s \) has type \( \sigma \) 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 \( s \triangleright t \), if (a) \( s = t \), (b) \( s = \lambda x . s' \) and \( s' \triangleright t \), (c) \( s = s_1 \triangleleft s_2 \) and \( s_1 \triangleright t \) or \( s_2 \triangleright t \), or (d) \( s = Z[s_1, \ldots, s_k] \) and some \( s_i \triangleright t \).
A meta-substitution is a type-preserving function $\gamma$ 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 $\lambda y_1 \ldots y_k : \sigma_1 \to \cdots \to \sigma_k \to \tau$. Let $\text{dom}(\gamma) = \{ x \in V \mid \gamma(x) \neq x \} \cup \{ Z \in M \mid \gamma(Z) \neq \lambda y_1 \ldots y_k Z[y_1, \ldots, y_k] \}$. (the domain of $\gamma$). We let $[b_1 := s_1, \ldots, b_n := s_n]$ be the meta-substitution $\gamma$ with $\gamma(b_i) = s_i$ if $z \in V \setminus \{ b \}$, and $\gamma(Z) = \lambda y_1 \ldots y_k Z[y_1, \ldots, y_k]$ for $Z \in M \setminus \{ b \}$. A substitution is a meta-substitution mapping everything in its domain to terms. The result $s_\gamma$ of applying a meta-substitution $\gamma$ to a meta-term $s$ is obtained recursively (with implicit $\alpha$-conversion):

$$
x\gamma = \gamma(x) \text{ if } x \in V \quad (s \ t)\gamma = (s\gamma) \ (t\gamma)
$$

$$
\text{if } f \in F \quad (\lambda x.s)\gamma = \lambda x. (s\gamma) \quad \text{if } \gamma(x) = x \land x \notin \text{FV}(s\gamma)
$$

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

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

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

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

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

$$
\text{rec } 0 \text{ K } F \text{ G } \Rightarrow \text{ K }, \quad \text{rec } (s \ X) \text{ K } F \text{ G } \Rightarrow \text{ F X (rec X K F G) },
$$

$$
\text{rec } (\text{lim } H) \text{ K } F \text{ G } \Rightarrow \text{ G H (}\lambda m. \text{rec} (H \ m) \text{ K } F \text{ G )}
$$

Then $\text{rec } (s \ 0) \text{ zero } (\lambda v.z) (\lambda x.z) \Rightarrow_\beta (\lambda v.z) (\lambda x.z) 0$ $\Rightarrow_\beta 0$ $\Rightarrow_\beta 0$.

**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 $\triangleright^S$ on the set of sorts (base types) $S$ with well-founded strict part $\triangleright_S := \triangleright^S \setminus \triangleright_S$. For $\sigma \equiv \sigma_1 \to \cdots \to \sigma_m \to \kappa$ (with $\kappa \in S$) and sort $\iota$, let $\iota \triangleright_S \sigma$ if $\iota \triangleright \sigma$ and each $\iota \triangleright_S \sigma_i$, and let $\iota \triangleright^S \sigma$ if $\iota \triangleright_S \sigma$ and each $\iota \triangleright^S \sigma_i$. (The relation $\iota \triangleright_S \sigma$ corresponds to “$\iota$ occurs only positively in $\sigma$” in [1, 4, 5].)

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

**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:

$\triangleright$ $s : \sigma \Rightarrow \tau$ is $\mathcal{R}$-computable iff $s \ t$ is $\mathcal{R}$-computable whenever $t : \sigma$ is $\mathcal{R}$-computable;

$\triangleright$ $s : i$ for a sort is $\mathcal{R}$-computable iff (1) $s$ is terminating under $\Rightarrow \cup \Rightarrow_I$ and (2) if $s \Rightarrow^* \mathcal{R}$ then $s_i$ is $\mathcal{R}$-computable for all $i \in \text{Acc}(f)$. Here, $f s_1 \cdots s_m \Rightarrow_I t_1 s_1 \cdots t_n$ if both sides have (possibly different) base types, $i \in \text{Acc}(f)$, 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 $\succeq^S$ such that $\text{ord} \succeq^S \text{nat}$. In Ex. 1, we then have $\text{ord} \succeq^S \text{nat} \rightarrow \text{ord}$. Therefore, $1 \in \text{Acc}(\lim)$, which gives $\lim H \subseteq \text{acc} H$.

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 $\mathcal{F}^s := \mathcal{F} \cup \{ f^s : \sigma | f : \sigma \in \mathcal{D} \}$. For a meta-term $s$, let $s^2 := f^s s_1 \cdots s_k$ if $s = f s_1 \cdots s_k$ with $f \in D$; let $s^2 := s$ otherwise. A DP is a pair $\ell \Rightarrow p$ where $\ell$ is a closed pattern $f \ell_1 \cdots \ell_m$, $p$ is a meta-term $g p_1 \cdots p_k$, and both $\ell$ and $p$ are $\beta$-normal and have (possibly different) base types.

The original static approaches define DPs as pairs $\ell \Rightarrow p^2$ with $\ell \Rightarrow r$ a rule and $p$ a subterm $g p_1 \cdots p_k$ 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 (\lambda-bound variables in functional programming, and our “meta-variables” free variables.)

Definition 6 (SDP). For a meta-term $s$, $\text{metafy}(s)$ denotes $s$ with all free variables replaced by corresponding fresh meta-variables. For an AFSM $(\mathcal{F}, \mathcal{R})$, $\text{SDP}(\mathcal{R}) = \{ \ell \Rightarrow \text{metafy}(p^2) | \ell \Rightarrow r \in \mathcal{R} \wedge r \succeq p \wedge \ell$ and $p$ have base types $\wedge$ and $p$ has the form $g p_1 \cdots p_k$ for some $g \in \mathcal{D} \}$.

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

Example 7. For Ex. 1, we obtain $\text{SDP}(\mathcal{R}) = \{ \text{rec}^2 (s X) K F G \Rightarrow \text{rec}^2 (s X) K F G, \text{rec}^2 (\lim H) K F G \Rightarrow \text{rec}^2 (H M) K F G \}$.

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 $[(p_0, s_0, t_0), (p_1, s_1, t_1), \ldots]$ where each $p_i \in \mathcal{P}$ and all $s_i, t_i$ are terms, such that for all $i$:

1. if $p_i = \ell_i \Rightarrow p_i$, then there exists a substitution $\gamma$ on domain $FMV(\ell_i) \cup FMV(p_i)$ such that $s_i = \ell_i \gamma$ and $t_i = p_i \gamma$; and
2. we can write $t_i = f u_1 \cdots u_n$ with $f \in \mathcal{F}^s$, $s_{i+1} = f w_1 \cdots w_n$ 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 $\succeq^S$ following Def. 2 such that:

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

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 AF-SM from Ex. 1 is AFP because of the sort ordering \( \text{ord} \succ^S \text{nat} \) (see also Ex. 4), yet it is not plain function passing following [14].

► Theorem 11. If \((F, R)\) is non-terminating and AFP, then there is an infinite minimal \((SDP(R), R)\)-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 \((P, R)\)-chain \([(l_0 \Rightarrow p_0, s_0, t_0), (l_1 \Rightarrow p_1, s_1, t_1), \ldots]\) is \(U\)-computable for a set of rules \(U\) if \(\Rightarrow_U \supseteq \Rightarrow_R\), for all \(i\) there exists a substitution \(\gamma_i\) with \(s_i = l_i\gamma_i\) and \(t_i = p_i\gamma_i\), and \((\lambda x_1 \ldots x_n)\gamma_i\) is \(U\)-computable for all \(v\) such that \(p_i \supseteq v\) and \(FV(v) = \{x_1, \ldots, x_n\}\).

► Theorem 13. (a) If an AF-SM \((F, R)\) is non-terminating and AFP, then there is an infinite \(R\)-computable \((SDP(R), R)\)-chain. (b) Every \(U\)-computable \((P, R)\)-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 AF-SMs \(R\) with infinite \(R\)-computable \((SDP(R), R)\)-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 \((P, R, m)\) with \(P\) a set of DPs, \(R\) a set of rules, and \(m \in \{\text{minimal, arbitrary}\} \cup \{\text{computable}_U \mid U\text{ a set of rules}\}\). A DP problem \((P, R, m)\) is finite if there exists no infinite \((P, R)\)-chain that is \(U\)-computable if \(m = \text{computable}_U\) or minimal if \(m = \text{minimal}\). For the different levels of permissiveness, we use a transitive-reflexive relation \(\succeq\) generated by \(\text{computable}_U \succeq \text{minimal} \succeq \text{arbitrary}\).

Thm. 13 now becomes: an AF-SM \((F, R)\) is terminating if (but not only if) it is AFP and \((SDP(R), R, \text{computable}_R)\) is finite. We add a flag value \(\text{computable}_R\) 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 $M = (\mathcal{P}_1 \cup \mathcal{P}_2, \mathcal{R}, \text{computable}_\mathcal{R})$ be a DP problem. A projection function $\nu$ maps meta-terms to natural numbers such that for all DPs $\ell \Rightarrow p \in \mathcal{P}_1 \cup \mathcal{P}_2$, the function $\varpi$ with $\varpi(\ell \ s_1 \ldots s_m) = s_{\nu(\ell)}$ is well-defined for $\ell$ and $p$. For meta-terms $s$ and $t$ of base types, we define $s \sqsupset t$ if $s \neq t$ and (a) $s \supseteq \text{acc} t$ or (b) there exists a meta-variable $Z$ with $s \supseteq \text{acc} Z[x_1, \ldots, x_k]$ and $t = Z[t_1, \ldots, t_k] \ s_1 \ldots s_n$. Then the processor $\text{Proc}_{\text{computable}}$ that maps $M$ to $((\mathcal{P}_2, \mathcal{R}, \text{computable}_\mathcal{R}))$ is sound if a projection function $\nu$ exists with $\varpi(\ell) \sqsupset \varpi(p)$ for all $\ell \Rightarrow p \in \mathcal{P}_1$ and $\varpi(\ell) = \varpi(p)$ for all $\ell \Rightarrow p \in \mathcal{P}_2$.

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

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