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 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 \( F \) of function symbols and \( V \) of variables, each symbol \( a \) equipped with a type \( σ \). We also fix a set \( M \), disjoint from \( F \) and \( V \), of meta-variables, each equipped with a type declaration \( [σ₁ × ⋯ × σₖ] \to τ \) (where \( τ \) and all \( σᵢ \) are simple types). Meta-terms are expressions \( s \) where \( s : σ \) can be derived for some type \( σ \) by the following clauses:

(V) \( x : σ \) if \( x : σ \in V \)
(Φ) \( s : σ = t : σ \) if \( s : σ \to τ \) and \( t : σ \)
(F) \( f : σ \) if \( f : σ \in F \)
(Λ) \( λx.s : σ \to τ \) if \( x : σ ∈ V \) and \( s : τ \)

(M) \( Z[s₁, \ldots, sₖ] : τ \) if \( Z : [σ₁ × ⋯ × σₖ] \to τ \in M \) and \( s₁ : σ₁, \ldots, sₖ : σₖ \)

Terms are meta-terms without meta-variables, so derived without clause (M). Patterns are meta-terms where all meta-variable occurrences have the form \( Z[x₁, \ldots, xₖ] \) with all \( xᵢ \) 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 (Φ) is left-associative; abstractions (Λ) extend as far to the right as possible. A meta-term \( s \) has type \( σ \) if \( s : σ \); it has base type if \( σ ∈ S \), 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) \( s = λx.s' \) and \( s' \triangleright t \), (c) \( s = s₁ s₂ \) and \( s₁ \triangleright t \) or \( s₂ \triangleright t \), or (d) \( s = Z[s₁, \ldots, sₖ] \) and some \( sᵢ \triangleright t \).
A meta-substitution is a type-preserving function $\gamma$ from variables and meta-variables to meta-terms; if $Z : \{\sigma_1 \times \cdots \times \sigma_k\} \rightarrow \tau$ then $\gamma(Z)$ has the form $\lambda y_1 \ldots y_l : \sigma_1 \rightarrow \ldots \rightarrow \sigma_k \rightarrow \tau$.

Let $\text{dom}(\gamma) = \{x \in V \mid \gamma(x) \neq \emptyset\} \cup \{Z \in M \mid \gamma(Z) \neq \emptyset\} \cup \{y_1, \ldots, y_k\}$ (the domain of $\gamma$). We let $[b_1 := s_1, \ldots, b_n := s_n]$ be the meta-substitution $\gamma$ with $\gamma(b_i) = s_i$, $\gamma(z) = z$ for $z \in V \setminus \{b\}$, and $\gamma(Z) = x y_1 \ldots y_l. Z[y_1, \ldots, y_k]$ 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):

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

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

$$Z[s_1, \ldots, s_k]_{\gamma} = t[x_1 := s_\gamma[y_1], \ldots, x_k := s_k]_\gamma \quad \text{if} \quad \gamma(Z) = \lambda x_1 \ldots x_k.t$$

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

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

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

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

Let $\mathcal{R}$ be:

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

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

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

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

For $f : \sigma_1 \rightarrow \cdots \rightarrow \sigma_m \rightarrow \iota \in F$, let $\text{Acc}(f) = \{i \mid 1 \leq i \leq m \land \iota \succeq^S \sigma_i\}$. For $x : \sigma_1 \rightarrow \cdots \rightarrow \sigma_m \rightarrow \iota \in V$, let $\text{Acc}(x) = \{i \mid 1 \leq i \leq m \land \sigma_i\}$ has the form $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \kappa$ for some $\kappa \in \mathbb{N}$ with $\iota \succeq^S \kappa$. We write $s \succeq^S_{\text{acc}} t$ if either $s = t$, or $s = \lambda x.s'$ and $s' \succeq^S_{\text{acc}} t$, or $s = a_{i_1} \cdots a_{i_n}$ with $a \in F \cup V$ and $s_i \succeq^S_{\text{acc}} t$ 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:

- $s : \sigma \rightarrow \tau$ is $\mathcal{R}$-computable iff $s t$ is $\mathcal{R}$-computable whenever $t : \sigma$ is $\mathcal{R}$-computable;
- $s : i$ for a sort is $\mathcal{R}$-computable if (1) $s$ is terminating under $\Rightarrow_{\mathcal{R}} \sqcup \Rightarrow_1$ and (2) if $s \Rightarrow_\mathcal{R} f s_1 \cdots s_m$ then $s_i$ is $\mathcal{R}$-computable for all $i \in \text{Acc}(f)$. Here, $f s_1 \cdots s_m \Rightarrow_1 f s_1 t_1 \cdots t_n$ 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].

\[ \text{Example 4.} \] 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, \( 1 \in \text{Acc}(\text{lim}) \), which gives \( \text{lim} H \succeq_{\text{acc}} H \).

\section{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:

\[ \text{Definition 5 (Marked symbols, DPs).} \] Define \( F^3 := F \cup \{ f^2 : \sigma \mid f : \sigma \in D \} \). For a meta-term \( s \), let \( s' := f^2 s_1 \cdots s_k \) if \( s = f s_1 \cdots s_k \) with \( f \in D \); let \( s' := s \) 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.)

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

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

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

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

\[ \text{Definition 8 (Dependency chain, minimal chain).} \] Let \( P \) be a set of DPs and \( R \) be a set of rules. A (finite or infinite) \((P, R)\)-dependency chain (or just \((P, R)\)-chain) is a sequence \([\{p_0, s_0, t_0\}, \{p_1, s_1, t_1\}, \ldots] \) where each \( p_i \in P \) 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 \( \text{FMV}(\ell_i) \cup \text{FMV}(p_i) \) 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 \( f \in F^3 \), \( s_{i+1} = f w_1 \cdots w_n \) and each \( u_j \Rightarrow_{R} w_j \).

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

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.

\[ \text{Definition 9 (Accessible function passing).} \] An AFSM \((F, R)\) 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 \( R \), i.e., they occur only with the maximum number of arguments permitted by their type;
- for all \( f \ell_1 \cdots \ell_m \Rightarrow r \in R \) and all \( Z \in \text{FMV}(r) \) there are some variables \( x_1, \ldots, x_k \) and some \( i \) such that \( \ell_i \succeq_{\text{acc}} Z[x_1, \ldots, x_k] \).

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.
Improving Static Dependency Pairs for Higher-Order Rewriting

The static DP framework follows the first-order DP framework [8], as an extendable framework 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 \textit{computable subterm criterion}, that needs the new flag.
Theorem 16 (Computable subterm criterion processor). Let \( M = (P_1 \cup P_2, R, \text{computable}_{\mathcal{R}}) \) be a DP problem. A projection function \( \nu \) maps meta-terms to natural numbers such that for all DPs \( \ell \Rightarrow p \in P_1 \cup P_2 \), the function \( \nu(f(s_1 \cdots s_m)) = s_{\nu(t)} \) is well-defined for \( \ell \) and \( p \). For meta-terms \( s \) and \( t \) of base types, we define \( s \sqsubseteq t \) if \( s \neq t \) and (a) \( s \sqsubseteq_{\text{acc}} t \) or (b) there exists a meta-variable \( Z \) with \( s \sqsubseteq_{\text{acc}} Z[x_1, \ldots, x_k] \) and \( t = Z[t_1, \ldots, t_k] s_1 \cdots s_n \). Then the processor \( \text{Proc}_{\text{compsub}} \) that maps \( M \) to \( \{(P_2, R, \text{computable}_{\mathcal{R}})\} \) is sound if a projection function \( \nu \) exists with \( \nu(\ell) \sqsubseteq \nu(p) \) for all \( \ell \Rightarrow p \in P_1 \) and \( \nu(\ell) = \nu(p) \) for all \( \ell \Rightarrow p \in P_2 \).

Example 17. \( R \) from Ex. 1 is terminating if \( (P, R, \text{computable}_{\mathcal{R}}) \) with \( P = \text{SDP}(R) \) is finite (see Ex. 7). Consider the projection function \( \nu \) with \( \nu(\text{rec}^2) = 1 \). As \( s X \sqsubseteq_{\text{acc}} X \) and \( \lim H \sqsubseteq_{\text{acc}} H \), we have \( s X \sqsubseteq X \) and \( \lim H \sqsubseteq H M \). So \( \text{Proc}_{\text{compsub}}(P, R, \text{computable}_{\mathcal{R}}) = \{(\emptyset, R, \text{computable}_{\mathcal{R}})\} \). As there are no DPs left, this implies termination of the original \( R \).

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