A static higher-order dependency pair framework (extended version)

Carsten Fuhs[1] and Cynthia Kop[2]

[1] Dept. of Comp. Sci. and Inf. Sys., Birkbeck, University of London, UK
[2] Dept. of Software Science, Radboud University Nijmegen, The Netherlands

Abstract We revisit the static dependency pair method for proving termination of higher-order term rewriting and extend it in a number of ways: (1) We introduce a new rewrite formalism designed for general applicability in termination proving of higher-order rewriting, Algebraic Functional Systems with Meta-variables. (2) We provide a syntactically checkable soundness criterion to make the method applicable to a large class of rewrite systems. (3) We propose a modular dependency pair *framework* for this higher-order setting. (4) We introduce a fine-grained notion of *formative* and *computable* chains to render the framework more powerful. (5) We formulate several existing and new termination proving techniques in the form of processors within our framework.
The framework has been implemented in the (fully automatic) higher-order termination tool WANDA.

1 Introduction

Term rewriting [3,48] is an important area of logic, with applications in many different areas of computer science [4,11,18,23,25,36,41]. *Higher-order* term rewriting – which extends the traditional *first-order* term rewriting with higher-order types and binders as in the λ-calculus – offers a formal foundation of functional programming and a tool for equational reasoning in higher-order logic. A key question in the analysis of both first- and higher-order term rewriting is *termination*; both for its own sake, and as part of confluence and equivalence analysis.

In first-order term rewriting, a hugely effective method for proving termination (both manually and automatically) is the *dependency pair (DP) approach* [2]. This approach has been extended to the *DP framework* [20,22], a highly modular methodology which new techniques for proving termination *and non-termination* can easily be plugged into in the form of *processors*.

In higher-order rewriting, two DP approaches with distinct costs and benefits are used: *dynamic* [45,31] and *static* [6,44,34,46,32,33] DPs. Dynamic DPs are more broadly applicable, yet static DPs often enable more powerful analysis techniques. Still, neither approach has the modularity and extendability of the DP framework, nor can they be used to prove non-termination. Also, these approaches consider different styles of higher-order rewriting, which means that for all results certain language features are not available.

In this paper, we address these issues for the *static* DP approach by extending it to a full higher-order *dependency pair framework* for both termination and non-termination analysis. For broad applicability, we introduce a new rewriting formalism, *AFSMs*, to capture several flavours of higher-order rewriting, including *AFSs* [26] (used in the annual Termination Competition [50]) and *pattern HRSs* [39,37] (used in the annual Confluence Competition [10]). To show the versatility and power of this methodology, we define various processors in the framework – both adaptations of existing processors from the literature and entirely new ones.

Detailed contributions. We reformulate the results of [6,44,34,46,32] into a DP framework for AFSMs. In doing so, we instantiate the applicability restriction of [32] by a very liberal syntactic condition, and add two new flags to track properties of DP problems: one completely new, one from an earlier work by the authors for the *first-order* DP framework [16]. We give eight *processors* for reasoning in our framework: four translations of techniques from static DP approaches, three techniques from first-order or dynamic DPs, and one completely new.

This is a *foundational* paper, focused on defining a general theoretical framework for higher-order termination analysis using dependency pairs rather than questions of implementation. We have, however, implemented most of these results in the fully automatic termination analysis tool WANDA [28].

Related Work. There is a vast body of work in the first-order setting regarding the DP approach [2] and framework [20,22,24]. We have drawn from the ideas in these works for the core structure of the higher-order framework, but have added some new features of our own and adapted results to the higher-order setting.

There is no true higher-order DP *framework* yet: both static and dynamic approaches actually lie halfway between the original "DP approach" of first-order rewriting and a full DP framework as in [20,22]. Most of these works [30,31,32,34,46] prove "non-loopingness" or "chain-freeness" of a set $\mathcal{P}$ of DPs through a number of theorems. Yet, there is no concept of *DP problems*, and the set $\mathcal{R}$ of rules cannot be altered. They also fix assumptions on dependency chains – such as minimality [34] or being "tagged" [31] – which frustrate extendability and are more naturally dealt with in a DP framework using flags.

The static DP approach for higher-order term rewriting is discussed in, e.g., [34,44,46]. The approach is limited to *plain function passing (PFP)* systems. The definition of PFP has been made more liberal in later papers, but always concerns the position of higher-order variables in the left-hand sides of rules. These works include non-pattern HRSs [34,46], which we do not consider, but do not employ formative rules or meta-variable conditions, or consider non-termination, which we do. Importantly, they do not consider strictly positive inductive types, which could be used to significantly broaden the PFP restriction. Such types *are* considered in an early paper which defines a variation of static higher-order dependency pairs [6] based on a computability closure [8,7]. However, this work carries different restrictions (e.g., DPs must be type-preserving and not introduce fresh variables) and considers only one analysis technique (reduction pairs).

Definitions of DP approaches for *functional programming* also exist [32,33], which consider applicative systems with ML-style polymorphism. These works also employ a much broader, semantic definition than PFP, which is actually more general than the syntactic restriction we propose here. However, like the static approaches for term rewriting, they do not truly exploit the computability [47] properties inherent in this restriction: it is only used for the initial generation of dependency pairs. In the present work, we will take advantage of our exact computability notion by introducing a `computable` flag that can be used by the computable subterm criterion processor (Thm. 63) to handle benchmark systems that would otherwise be beyond the reach of static DPs. Also in these works, formative rules, meta-variable conditions and non-termination are not considered.

Regarding *dynamic* DP approaches, a precursor of the present work is [31], which provides a halfway framework (methodology to prove "chain-freeness") for dynamic DPs, introduces a notion of formative rules, and briefly translates a basic form of static DPs to the same setting. Our formative *reductions* consider the shape of reductions rather than the rules they use, and they can be used as a flag in the framework to gain additional power in other processors. The adaptation of static DPs in [31] was very limited, and did not for instance consider strictly positive inductive types or rules of functional type.

For a more elaborate discussion of both static and dynamic DP approaches in the literature, we refer to [31] and the second author's PhD thesis [29].

Organisation of the paper. § 2 introduces higher-order rewriting using AFSMs and recapitulates computability. In § 3 we impose restrictions on the input AFSMs for which our framework is soundly applicable. In § 4 we define static DPs for AFSMs, and derive the key results on them. § 5 formulates the DP framework and a number of DP processors for existing and new termination proving techniques. § 6 concludes. Detailed proofs for all results in this paper (extending [17]) and an experimental evaluation are available in the appendix. In addition, many of the results have been informally published in the second author's PhD thesis [29].

2 Preliminaries

In this section, we first define our notation by introducing the AFSM formalism. Although not one of the standards of higher-order rewriting, AFSMs combine features from various forms of higher-order rewriting and can be seen as a form of IDTSs [5] which includes application. We will finish with a definition of *computability*, a technique often used for higher-order termination methods.

2.1 Higher-order term rewriting using AFSMs

Unlike first-order term rewriting, there is no single, unified approach to higher-order term rewriting, but rather a number of similar but not fully compatible systems aiming to combine term rewriting and typed λ-calculi. For generality, we will use *Algebraic Functional Systems with Meta-variables*: a formalism which admits translations from the main formats of higher-order term rewriting.

Definition 1 (Simple types). *We fix a set $\mathcal{S}$ of* sorts. *All sorts are simple types, and if σ, τ are simple types, then so is $\sigma \to \tau$.*

We let $\rightarrow$ be right-associative. Note that all types have a unique representation in the form $\sigma_1 \rightarrow \ldots \rightarrow \sigma_m \rightarrow \iota$ with $\iota \in \mathcal{S}$.

Definition 2 (Terms and meta-terms). *We fix disjoint sets $\mathcal{F}$ of* function symbols*, $\mathcal{V}$ of* variables *and $\mathcal{M}$ of* meta-variables*, each symbol equipped with a type. Each meta-variable is additionally equipped with a natural number. We assume that both $\mathcal{V}$ and $\mathcal{M}$ contain infinitely many symbols of all types. The set $\mathcal{T}(\mathcal{F},\mathcal{V})$ of* terms *over $\mathcal{F},\mathcal{V}$ consists of 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* $s : \tau$

Meta-terms *are expressions whose type can be derived by those clauses and:*

(M) $Z\langle s_1,\ldots,s_k\rangle : \sigma_{k+1} \rightarrow \ldots \rightarrow \sigma_m \rightarrow \iota$
if $Z : (\sigma_1 \rightarrow \ldots \rightarrow \sigma_k \rightarrow \ldots \rightarrow \sigma_m \rightarrow \iota,\ k) \in \mathcal{M}$ *and* $s_1 : \sigma_1,\ldots,s_k : \sigma_k$

The λ binds variables as in the λ-calculus; unbound variables are called free*, and $FV(s)$ is the set of free variables in s. Meta-variables cannot be bound; we write $FMV(s)$ for the set of meta-variables occurring in s. A meta-term s is called* closed *if $FV(s) = \emptyset$ (even if $FMV(s) \neq \emptyset$). 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 $\sigma \in \mathcal{S}$. We define* $\mathsf{head}(s) = \mathsf{head}(s_1)$ *if $s = s_1\ s_2$, and* $\mathsf{head}(s) = s$ *otherwise.*

A (meta-)term s has a sub-(meta-)term *t, notation $s \trianglerighteq t$, if either $s = t$ or $s \triangleright t$, where $s \triangleright t$ if (a) $s = \lambda x.s'$ and $s' \trianglerighteq t$, (b) $s = s_1\ s_2$ and $s_2 \trianglerighteq t$ or (c) $s = s_1\ s_2$ and $s_1 \trianglerighteq t$. A (meta-)term s has a* fully applied sub-(meta-)term *t, notation $s \blacktriangleright\!\!\!\!\underline{\ \ } t$, if either $s = t$ or $s \blacktriangleright t$, where $s \blacktriangleright t$ if (a) $s = \lambda x.s'$ and $s' \blacktriangleright\!\!\!\!\underline{\ \ } t$, (b) $s = s_1\ s_2$ and $s_2 \blacktriangleright\!\!\!\!\underline{\ \ } t$ or (c) $s = s_1\ s_2$ and $s_1 \blacktriangleright t$ (so if $s = x\ s_1\ s_2$, then x and $x\ s_1$ are not fully applied subterms, but s and both s_1 and s_2 are).*

For $Z : (\sigma, k) \in \mathcal{M}$, we call k the arity *of Z, notation $arity(Z)$.*

Clearly, all fully applied subterms are subterms, but not all subterms are fully applied. Every term s has a form $t\ s_1 \cdots s_n$ with $n \geq 0$ and $t = \mathsf{head}(s)$ a variable, function symbol, or abstraction; in meta-terms t may also be a meta-variable application $F\langle s_1,\ldots,s_k\rangle$. *Terms* are the objects that we will rewrite; *meta-terms* are used to define rewrite rules. Note that all our terms (and meta-terms) are, by definition, well-typed. For rewriting, we will employ *patterns*:

Definition 3 (Patterns). *A meta-term is a* pattern *if it has one of the forms $Z\langle x_1,\ldots,x_k\rangle$ with all x_i distinct variables; $\lambda x.\ell$ with $x \in \mathcal{V}$ and ℓ a pattern; or $a\ \ell_1 \cdots \ell_n$ with $a \in \mathcal{F} \cup \mathcal{V}$ and all ℓ_i patterns ($n \geq 0$).*

In rewrite rules, we will use meta-variables for *matching* and variables only with *binders*. In terms, variables can occur both free and bound, and meta-variables cannot occur. Meta-variables originate in very early forms of higher-order rewriting (e.g., [1,27]), but have also been used in later formalisms (e.g., [8]). They strike a balance between matching modulo β and syntactic matching. By using meta-variables, we obtain the same expressive power as with Miller patterns [37], but do so without including a reversed β-reduction as part of matching.

Notational conventions: We will use x, y, z for variables, X, Y, Z for meta-variables, b for symbols that could be variables or meta-variables, $\mathtt{f}, \mathtt{g}, \mathtt{h}$ or more suggestive notation for function symbols, and s, t, u, v, q, w for (meta-)terms. Types are denoted σ, τ, and ι, κ are sorts. We will regularly overload notation and write $x \in \mathcal{V}$, $\mathtt{f} \in \mathcal{F}$ or $Z \in \mathcal{M}$ without stating a type (or minimal arity). For meta-terms $Z\langle\rangle$ we will usually omit the brackets, writing just Z.

Definition 4 (Substitution). *A* meta-substitution *is a type-preserving function γ from variables and meta-variables to meta-terms. Let the* domain *of γ be given by: $\mathtt{dom}(\gamma) = \{(x : \sigma) \in \mathcal{V} \mid \gamma(x) \neq x\} \cup \{(Z : (\sigma, k)) \in \mathcal{M} \mid \gamma(Z) \neq \lambda y_1 \dots y_k.Z\langle y_1, \dots, y_k\rangle\}$; this domain is allowed to be infinite. We let $[b_1 := s_1, \dots, b_n := s_n]$ denote the meta-substitution γ with $\gamma(b_i) = s_i$ and $\gamma(z) = z$ for $(z : \sigma) \in \mathcal{V} \setminus \{b_1, \dots, b_n\}$, and $\gamma(Z) = \lambda y_1 \dots y_k.Z\langle y_1, \dots, y_k\rangle$ for $(Z : (\sigma, k)) \in \mathcal{M} \setminus \{b_1, \dots, b_n\}$. We assume there are infinitely many variables x of all types such that (a) $x \notin \mathtt{dom}(\gamma)$ and (b) for all $b \in \mathtt{dom}(\gamma)$: $x \notin FV(\gamma(b))$.*

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 term s is obtained by:*

$$x\gamma = \gamma(x) \text{ if } x \in \mathcal{V} \qquad (s\ t)\gamma = (s\gamma)\ (t\gamma)$$
$$\mathtt{f}\gamma = \mathtt{f} \text{ if } \mathtt{f} \in \mathcal{F} \qquad (\lambda x.s)\gamma = \lambda x.(s\gamma) \text{ if } \gamma(x) = x \wedge x \notin \bigcup_{y \in \mathtt{dom}(\gamma)} FV(\gamma(y))$$

For meta-terms, the result $s\gamma$ is obtained by the clauses above and:

$$Z\langle s_1, \dots, s_k\rangle\gamma = \gamma(Z)\langle s_1\gamma, \dots, s_k\gamma\rangle \quad \text{if } Z \notin \mathtt{dom}(\gamma)$$
$$Z\langle s_1, \dots, s_k\rangle\gamma = \gamma(Z)\langle\langle s_1\gamma, \dots, s_k\gamma\rangle\rangle \quad \text{if } Z \in \mathtt{dom}(\gamma)$$
$$(\lambda x_1 \dots x_k.s)\langle\langle t_1, \dots, t_k\rangle\rangle = s[x_1 := t_1, \dots, x_k := t_k]$$
$$(\lambda x_1 \dots x_n.s)\langle\langle t_1, \dots, t_k\rangle\rangle = s[x_1 := t_1, \dots, x_n := t_n]\ t_{n+1} \cdots t_k \quad \text{if } n < k \text{ and } s \text{ is not an abstraction}$$

Note that for fixed k, any term has exactly one of the two forms above ($\lambda x_1 \dots x_n.s$ with $n < k$ and s not an abstraction, or $\lambda x_1 \dots x_k.s$).

Essentially, applying a meta-substitution that has meta-variables in its domain combines a substitution with (possibly several) β-steps. For example, we have that: $\mathtt{deriv}\ (\lambda x.\mathtt{sin}\ (F\langle x\rangle))[F := \lambda y.\mathtt{plus}\ y\ x]$ equals $\mathtt{deriv}\ (\lambda z.\mathtt{sin}\ (\mathtt{plus}\ z\ x))$. We also have: $X\langle \mathtt{0}, \mathtt{nil}\rangle[X := \lambda x.\mathtt{map}\ (\lambda y.x)]$ equals $\mathtt{map}\ (\lambda y.\mathtt{0})\ \mathtt{nil}$.

Definition 5 (Rules and rewriting). *Let $\mathcal{F}, \mathcal{V}, \mathcal{M}$ be fixed sets of function symbols, variables and meta-variables respectively. A* rule *is a pair $\ell \Rightarrow r$ of closed meta-terms of the same type such that ℓ is 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 $\mathcal{R}$ defines a rewrite relation $\Rightarrow_{\mathcal{R}}$ as the smallest monotonic relation on terms which includes:*

(Rule) $\ell\delta \Rightarrow_{\mathcal{R}} r\delta$ *if $\ell \Rightarrow r \in \mathcal{R}$ and $\mathtt{dom}(\delta) = FMV(\ell)$*
(Beta) $(\lambda x.s)\ t \Rightarrow_{\mathcal{R}} s[x := t]$

We say $s \Rightarrow_\beta t$ if $s \Rightarrow_{\mathcal{R}} t$ is derived using a (Beta) step. A term s is terminating *under $\Rightarrow_{\mathcal{R}}$ if there is no infinite reduction $s = s_0 \Rightarrow_{\mathcal{R}} s_1 \Rightarrow_{\mathcal{R}} \dots$, is* in normal form *if there is no t such that $s \Rightarrow_{\mathcal{R}} t$, and is* β-normal *if there is no t with $s \Rightarrow_\beta t$. Note that we are allowed to reduce at any position of a term, even below a λ. The relation $\Rightarrow_{\mathcal{R}}$ is terminating if all terms over $\mathcal{F}, \mathcal{V}$ are terminating. The set $\mathcal{D} \subseteq \mathcal{F}$ of* defined symbols *consists of those $(\mathtt{f} : \sigma) \in \mathcal{F}$ such that a rule $\mathtt{f}\ \ell_1 \cdots \ell_n \Rightarrow r$ exists; all other symbols are called* constructors.

Note that $\mathcal{R}$ is allowed to be infinite, which is useful for instance to model polymorphic systems. Also, right-hand sides of rules do not have to be in β-normal form. While this is rarely used in practical examples, non-β-normal rules may arise through transformations, and we lose nothing by allowing them.

Example 6. Let $\mathcal{F} \supseteq \{\mathtt{0} : \mathtt{nat},\ \mathtt{s} : \mathtt{nat} \to \mathtt{nat},\ \mathtt{nil} : \mathtt{list}, \mathtt{cons} : \mathtt{nat} \to \mathtt{list} \to \mathtt{list},\ \mathtt{map} : (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{list} \to \mathtt{list}\}$ and consider the following rules $\mathcal{R}$:

$$\begin{array}{c} \mathtt{map}\ (\lambda x.Z\langle x\rangle)\ \mathtt{nil} \Rightarrow \mathtt{nil} \\ \mathtt{map}\ (\lambda x.Z\langle x\rangle)\ (\mathtt{cons}\ H\ T) \Rightarrow \mathtt{cons}\ Z\langle H\rangle\ (\mathtt{map}\ (\lambda x.Z\langle x\rangle)\ T) \end{array}$$

Then $\mathtt{map}\ (\lambda y.\mathtt{0})\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{nil}) \Rightarrow_{\mathcal{R}} \mathtt{cons}\ \mathtt{0}\ (\mathtt{map}\ (\lambda y.\mathtt{0})\ \mathtt{nil}) \Rightarrow_{\mathcal{R}} \mathtt{cons}\ \mathtt{0}\ \mathtt{nil}$. Note that the bound variable y does not need to occur in the body of $\lambda y.\mathtt{0}$ to match $\lambda x.Z\langle x\rangle$. However, a term like $\mathtt{map}\ \mathtt{s}\ (\mathtt{cons}\ \mathtt{0}\ \mathtt{nil})$ *cannot* be reduced, because $\mathtt{s}$ does not instantiate $\lambda x.Z\langle x\rangle$. We could alternatively consider the rules:

$$\begin{array}{c} \mathtt{map}\ Z\ \mathtt{nil} \Rightarrow \mathtt{nil} \\ \mathtt{map}\ Z\ (\mathtt{cons}\ H\ T) \Rightarrow \mathtt{cons}\ (Z\ H)\ (\mathtt{map}\ Z\ T) \end{array}$$

Where the system before had $(Z : (\mathtt{nat} \to \mathtt{nat}, 1)) \in \mathcal{M}$, here we assume $(Z : (\mathtt{nat} \to \mathtt{nat}, 0)) \in \mathcal{M}$. Thus, rather than meta-variable application $Z\langle H\rangle$ we use explicit application $Z\ H$. Then $\mathtt{map}\ \mathtt{s}\ (\mathtt{cons}\ \mathtt{0}\ \mathtt{nil}) \Rightarrow_{\mathcal{R}} \mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ (\mathtt{map}\ \mathtt{s}\ \mathtt{nil})$. However, we will often need explicit β-reductions; e.g., $\mathtt{map}\ (\lambda y.\mathtt{0})\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{nil}) \Rightarrow_{\mathcal{R}} \mathtt{cons}\ ((\lambda y.\mathtt{0})\ (\mathtt{s}\ \mathtt{0}))\ (\mathtt{map}\ (\lambda y.\mathtt{0})\ \mathtt{nil}) \Rightarrow_{\beta} \mathtt{cons}\ \mathtt{0}\ (\mathtt{map}\ (\lambda y.\mathtt{0})\ \mathtt{nil})$.

Definition 7 (AFSM). *An* AFSM *is a tuple $(\mathcal{F}, \mathcal{V}, \mathcal{M}, \mathcal{R})$ of a signature and a set of rules built from meta-terms over $\mathcal{F}, \mathcal{V}, \mathcal{M}$; as types of relevant variables and meta-variables can always be derived from context, we will typically just refer to the AFSM $(\mathcal{F}, \mathcal{R})$. An AFSM implicitly defines the abstract reduction system $(\mathcal{T}(\mathcal{F}, \mathcal{V}), \Rightarrow_{\mathcal{R}})$: a set of terms and a rewrite relation on this set. An AFSM is terminating if $\Rightarrow_{\mathcal{R}}$ is terminating (on all terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$).*

Discussion: The two most common formalisms in termination analysis of higher-order rewriting are *algebraic functional systems* [26] (AFSs) and *higher-order rewriting systems* [39,37] (HRSs). AFSs are very similar to our AFSMs, but use variables for matching rather than meta-variables; this is trivially translated to the AFSM format, giving rules where all meta-variables have arity 0, like the "alternative" rules in Ex. 6. HRSs use matching modulo β/η, but the common restriction of *pattern HRSs* can be directly translated into AFSMs, provided terms are β-normalised after every reduction step. Even without this β-normalisation step, termination of the obtained AFSM implies termination of the original HRS; for second-order systems, termination is equivalent. AFSMs can also naturally encode CRSs [27] and several applicative systems (cf. [29, Chapter 3]).

Example 8 (Ordinal recursion). A running example is the AFSM $(\mathcal{F}, \mathcal{R})$ with $\mathcal{F} \supseteq \{\mathtt{0} : \mathtt{ord},\ \mathtt{s} : \mathtt{ord} \to \mathtt{ord}, \mathtt{lim} : (\mathtt{nat} \to \mathtt{ord}) \to \mathtt{ord},\ \mathtt{rec} : \mathtt{ord} \to \mathtt{nat} \to (\mathtt{ord} \to \mathtt{nat} \to \mathtt{nat}) \to ((\mathtt{nat} \to \mathtt{ord}) \to (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}) \to \mathtt{nat}\}$ and $\mathcal{R}$ given below. As all meta-variables have arity 0, this can be seen as an AFS.

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

Observant readers may notice that by the given constructors, the type `nat` in Ex. 8 is not inhabited. However, as the given symbols are only a subset of $\mathcal{F}$, additional symbols (such as constructors for the `nat` type) may be included. The presence of additional function symbols does not affect termination of AFSMs:

Theorem 9 (Invariance of termination under signature extensions). *For an AFSM $(\mathcal{F}, \mathcal{R})$ with $\mathcal{F}$ at most countably infinite, let $\mathtt{funs}(\mathcal{R}) \subseteq \mathcal{F}$ be the set of function symbols occurring in some rule of $\mathcal{R}$. Then $(\mathcal{T}(\mathcal{F}, \mathcal{V}), \Rightarrow_{\mathcal{R}})$ is terminating if and only if $(\mathcal{T}(\mathtt{funs}(\mathcal{R}), \mathcal{V}), \Rightarrow_{\mathcal{R}})$ is terminating.*

Proof. Trivial by replacing all function symbols in $\mathcal{F} \setminus \mathtt{funs}(\mathcal{R})$ by corresponding variables of the same type. $\square$

Therefore, we will typically only state the types of symbols occurring in the rules, but may safely assume that infinitely many symbols of all types are present (which for instance allows us to select unused constructors in some proofs).

2.2 Computability

A common technique in higher-order termination is Tait and Girard's *computability* notion [47]. There are several ways to define computability predicates; here we follow, e.g., [5,8,9,7] in considering *accessible meta-terms* using strictly positive inductive types. The definition presented below is adapted from these works, both to account for the altered formalism and to introduce (and obtain termination of) a relation $\Rrightarrow_C$ that we will use in the "computable subterm criterion processor" of Thm. 63 (a termination criterion that allows us to handle systems that would otherwise be beyond the reach of static DPs). This allows for a minimal presentation that avoids the use of ordinals that would otherwise be needed to obtain $\Rrightarrow_C$ (see, e.g., [9,7]).

To define computability, we use the notion of an *RC-set*:

Definition 10. *A* set of reducibility candidates*, or* RC-set*, for a rewrite relation $\Rightarrow_{\mathcal{R}}$ of an AFSM is a set I of base-type terms s such that: every term in I is terminating under $\Rightarrow_{\mathcal{R}}$; I is closed under $\Rightarrow_{\mathcal{R}}$ (so if $s \in I$ and $s \Rightarrow_{\mathcal{R}} t$ then $t \in I$); if $s = x\ s_1 \cdots s_n$ with $x \in \mathcal{V}$ or $s = (\lambda x.u)\ s_0 \cdots s_n$ with $n \geq 0$, and for all t with $s \Rightarrow_{\mathcal{R}} t$ we have $t \in I$, then $s \in I$ (for any $u, s_0, \ldots, s_n \in \mathcal{T}(\mathcal{F}, \mathcal{V})$).*

We define I-computability for an RC-set I by induction on types. For $s \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we say that s is I-computable if either s is of base type and $s \in I$; or $s : \sigma \to \tau$ and for all $t : \sigma$ that are I-computable, $s\ t$ is I-computable.

The traditional notion of computability is obtained by taking for I the set of all terminating base-type terms. Then, a term s is computable if and only if (a) s has base type and is terminating; or (b) $s : \sigma \to \tau$ and for all computable $t : \sigma$ the term $s\ t$ is computable. This choice is simple but, for reasoning, not ideal: we do not have a property like: "if $\mathtt{f}\ s_1 \cdots s_n$ is computable then so is each s_i". Such a property would be valuable to have for generalising termination proofs from first-order to higher-order rewriting, as it allows us to use computability where the first-order proof uses termination. While it is not possible to define a computability notion with this property alongside case (b) (as such a notion would not be well-founded), we can come *close* to this property by choosing a different set for I. To define this set, we will use the notion of *accessible arguments*, which is used for the same purpose also in the *General Schema* [8], the *Computability Path Ordering* [9], and the *Computability Closure* [7].

Definition 11 (Accessible arguments). *We fix a quasi-ordering $\succeq^{\mathcal{S}}$ on $\mathcal{S}$ with well-founded strict part $\succ^{\mathcal{S}} := \succeq^{\mathcal{S}} \setminus \preceq^{\mathcal{S}}$.*[3] *For a type $\sigma \equiv \sigma_1 \to \ldots \to \sigma_m \to \kappa$ (with $\kappa \in \mathcal{S}$) and sort ι, let $\iota \succeq^{\mathcal{S}}_{+} \sigma$ if $\iota \succeq^{\mathcal{S}} \kappa$ and $\iota \succ^{\mathcal{S}}_{-} \sigma_i$ for all i, and let $\iota \succ^{\mathcal{S}}_{-} \sigma$ if $\iota \succ^{\mathcal{S}} \kappa$ and $\iota \succeq^{\mathcal{S}}_{+} \sigma_i$ for all i.*[4]

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 $\tau_1 \to \ldots \to \tau_n \to \kappa$ 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 $i \in Acc(a)$ with $a \notin FV(s_i)$.

With this definition, we will be able to define a set C such that, roughly, s is C-computable if and only if (a) $s : \sigma \to \tau$ and $s\ t$ is C-computable for all C-computable t, or (b) s has base type, is terminating, and if $s = \mathtt{f}\ s_1 \cdots s_m$ then s_i is C-computable for all *accessible* i (see Thm. 13 below). The reason that $Acc(x)$ for $x \in \mathcal{V}$ is different is proof-technical: computability of $\lambda x.x\ s_1 \cdots s_m$ implies the computability of more arguments s_i than computability of $\mathtt{f}\ s_1 \cdots s_m$ does, since x can be instantiated by anything.

Example 12. Consider a quasi-ordering $\succeq^{\mathcal{S}}$ such that $\mathtt{ord} \succ^{\mathcal{S}} \mathtt{nat}$. In Ex. 8, we then have $\mathtt{ord} \succeq^{\mathcal{S}}_{+} \mathtt{nat} \to \mathtt{ord}$. Thus, $1 \in Acc(\mathtt{lim})$, which gives $\mathtt{lim}\ H \trianglerighteq_{\mathtt{acc}} H$.

Theorem 13. *Let $(\mathcal{F}, \mathcal{R})$ be an AFSM. Let $\mathtt{f}\ s_1 \cdots s_m \Rrightarrow_I s_i\ t_1 \cdots t_n$ if both sides have base type, $i \in Acc(\mathtt{f})$, and all t_j are I-computable. There is an RC-set C such that $C = \{s \in \mathcal{T}(\mathcal{F}, \mathcal{V}) \mid s$ has base type $\wedge$ s is terminating under $\Rightarrow_{\mathcal{R}} \cup \Rrightarrow_C \wedge$ if $s \Rightarrow^*_{\mathcal{R}} \mathtt{f}\ s_1 \cdots s_m$ then s_i is C-computable for all $i \in Acc(\mathtt{f})\}$.*

Proof (sketch). Note that we cannot *define* C as this set, as the set relies on the notion of C-computability. However, we *can* define C as the fixpoint of a monotone function operating on RC-sets. This follows the proof in, e.g., [8,9]. □

The full proof (for the definitions in this paper) is available in Appendix A.

3 Restrictions

The termination methodology in this paper is restricted to AFSMs that satisfy certain limitations: they must be *properly applied* (a restriction on the number of terms each function symbol is applied to) and *accessible function passing* (a restriction on the positions of variables of a functional type in the left-hand sides of rules). Both are syntactic restrictions that are easily checked by a computer (mostly; the latter requires a search for a sort ordering, but this is typically easy).

[3] Well-foundedness is immediate if $\mathcal{S}$ is finite, but we have not imposed that requirement.
[4] Here $\iota \succeq^{\mathcal{S}}_{+} \sigma$ corresponds to "ι occurs only positively in σ" in [5,8,9].

3.1 Properly applied AFSMs

In *properly applied AFSMs*, function symbols are assigned a certain, minimal number of arguments that they must always be applied to.

Definition 14. *An AFSM $(\mathcal{F}, \mathcal{R})$ is* properly applied *if for every $\mathtt{f} \in \mathcal{D}$ there exists an integer k such that for all rules $\ell \Rightarrow r \in \mathcal{R}$: (1) if $\ell = \mathtt{f}\ \ell_1 \cdots \ell_n$ then $n = k$; and (2) if $r \trianglerighteq \mathtt{f}\ r_1 \cdots r_n$ then $n \geq k$. We denote $minar(\mathtt{f}) = k$.*

That is, every occurrence of a function symbol in the *right-hand* side of a rule has at least as many arguments as the occurrences in the *left-hand* sides of rules. This means that partially applied functions are often not allowed: an AFSM with rules such as $\mathtt{double}\ X \Rightarrow \mathtt{plus}\ X\ X$ and $\mathtt{doublelist}\ L \Rightarrow \mathtt{map}\ \mathtt{double}\ L$ is not properly applied, because $\mathtt{double}$ is applied to one argument in the left-hand side of some rule, and to zero in the right-hand side of another.

This restriction is not as severe as it may initially seem since partial applications can be replaced by λ-abstractions; e.g., the rules above can be made properly applied by replacing the second rule by: $\mathtt{doublelist}\ L \Rightarrow \mathtt{map}\ (\lambda x.\mathtt{double}\ x)\ L$. By using η-expansion, we can transform any AFSM to satisfy this restriction:

Definition 15 ($\mathcal{R}^{\uparrow}$). *Given a set of rules $\mathcal{R}$, let their* η-expansion *be given by $\mathcal{R}^{\uparrow} = \{(\ell\ Z_1 \cdots Z_m){\uparrow}^{\eta} \Rightarrow (r\ Z_1 \cdots Z_m){\uparrow}^{\eta} \mid \ell \Rightarrow r \in \mathcal{R}$ with $r : \sigma_1 \to \ldots \to \sigma_m \to \iota$, $\iota \in \mathcal{S}$, and $Z_1, \ldots, Z_m$ fresh meta-variables$\}$, where*

- *$s{\uparrow}^{\eta} = \lambda x_1 \ldots x_m.\overline{s}\ (x_1{\uparrow}^{\eta}) \cdots (x_m{\uparrow}^{\eta})$ if s is an application or element of $\mathcal{V} \cup \mathcal{F}$, and $s{\uparrow}^{\eta} = \overline{s}$ otherwise;*
- *$\overline{\mathtt{f}} = \mathtt{f}$ for $\mathtt{f} \in \mathcal{F}$ and $\overline{x} = x$ for $x \in \mathcal{V}$, while $\overline{Z\langle s_1, \ldots, s_k\rangle} = Z\langle \overline{s_1}, \ldots, \overline{s_k}\rangle$ and $\overline{(\lambda x.s)} = \lambda x.(s{\uparrow}^{\eta})$ and $\overline{s_1\ s_2} = \overline{s_1}\ (s_2{\uparrow}^{\eta})$.*

Note that $\ell{\uparrow}^{\eta}$ is a pattern if ℓ is. By [29, Thm. 2.16], a relation $\Rightarrow_{\mathcal{R}}$ is terminating if $\Rightarrow_{\mathcal{R}^{\uparrow}}$ is terminating, which allows us to transpose any methods to prove termination of properly applied AFSMs to all AFSMs.

However, there is a caveat: this transformation can introduce non-termination in some special cases, e.g., the terminating rule $\mathtt{f}\ X \Rightarrow \mathtt{g}\ \mathtt{f}$ with $\mathtt{f} : \mathtt{o} \to \mathtt{o}$ and $\mathtt{g} : (\mathtt{o} \to \mathtt{o}) \to \mathtt{o}$, whose η-expansion $\mathtt{f}\ X \Rightarrow \mathtt{g}\ (\lambda x.(\mathtt{f}\ x))$ is non-terminating. Thus, for a properly applied AFSM the methods in this paper apply directly. For an AFSM that is not properly applied, we can use the methods to prove *termination* (but not non-termination) by first η-expanding the rules. Of course, if this analysis leads to a *counterexample* for termination, we may still be able to verify whether this counterexample applies in the original, untransformed AFSM.

Example 16. Both AFSMs in Ex. 6 and the AFSM in Ex. 8 are properly applied.

Example 17. Consider an AFSM $(\mathcal{F}, \mathcal{R})$ with $\mathcal{F} \supseteq \{\mathtt{sin}, \mathtt{cos} : \mathtt{real} \to \mathtt{real}$, $\mathtt{times} : \mathtt{real} \to \mathtt{real} \to \mathtt{real}$, $\mathtt{deriv} : (\mathtt{real} \to \mathtt{real}) \to \mathtt{real} \to \mathtt{real}\}$ and $\mathcal{R} = \{\mathtt{deriv}\ (\lambda x.\mathtt{sin}\ F\langle x\rangle) \Rightarrow \lambda y.\mathtt{times}\ (\mathtt{deriv}\ (\lambda x.F\langle x\rangle)\ y)\ (\mathtt{cos}\ F\langle y\rangle)\}$. Although the one rule has a functional output type ($\mathtt{real} \to \mathtt{real}$), this AFSM is properly applied, with $\mathtt{deriv}$ having always at least 1 argument. Therefore, we do not need to use $\mathcal{R}^{\uparrow}$. However, if $\mathcal{R}$ were to additionally include some rules that did not satisfy the restriction (such as the `double` and `doublelist` rules above), then η-expanding *all* rules, including this one, would be necessary. We have: $\mathcal{R}^{\uparrow} = \{\mathtt{deriv}\ (\lambda x.\mathtt{sin}\ F\langle x\rangle)\ Y \Rightarrow (\lambda y.\mathtt{times}\ (\mathtt{deriv}\ (\lambda x.F\langle x\rangle)\ y)\ (\mathtt{cos}\ F\langle y\rangle))\ Y\}$. Note that the right-hand side of the η-expanded `deriv` rule is not β-normal.

3.2 Accessible Function Passing AFSMs

In *accessible function passing* AFSMs, variables of functional type may not occur at arbitrary places in the left-hand sides of rules: their positions are restricted using the sort ordering $\succeq^{\mathcal{S}}$ and accessibility relation $\trianglerighteq_{\mathtt{acc}}$ from Def. 11.

Definition 18 (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. 11 such that: for all* $\mathtt{f}\ \ell_1 \cdots \ell_n \Rightarrow r \in \mathcal{R}$ *and all* $Z \in FMV(r)$*: there are variables* $x_1, \dots, x_k$ *and some* i *such that* $\ell_i \trianglerighteq_{\mathtt{acc}} Z\langle x_1, \dots, x_k\rangle$*.*

The key idea of this definition is that computability of each ℓ_i implies computability of all meta-variables in r. This excludes cases like Example 20 below. Many common examples satisfy this restriction, including those we saw before:

Example 19. Both systems from Ex. 6 are AFP: choosing the sort ordering $\succeq^{\mathcal{S}}$ that equates `nat` and `list`, we indeed have $\mathtt{cons}\ H\ T \trianglerighteq_{\mathtt{acc}} H$ and $\mathtt{cons}\ H\ T \trianglerighteq_{\mathtt{acc}} T$ (as $Acc(\mathtt{cons}) = \{1, 2\}$) and both $\lambda x.Z\langle x\rangle \trianglerighteq_{\mathtt{acc}} Z\langle x\rangle$ and $Z \trianglerighteq_{\mathtt{acc}} Z$. The AFSM from Ex. 8 is AFP because we can choose $\mathtt{ord} \succ^{\mathcal{S}} \mathtt{nat}$ and have $\mathtt{lim}\ H \trianglerighteq_{\mathtt{acc}} H$ following Ex. 12 (and also $\mathtt{s}\ X \trianglerighteq_{\mathtt{acc}} X$ and $K \trianglerighteq_{\mathtt{acc}} K$, $F \trianglerighteq_{\mathtt{acc}} F$, $G \trianglerighteq_{\mathtt{acc}} G$). The AFSM from Ex. 17 is AFP, because $\lambda x.\mathtt{sin}\ F\langle x\rangle \trianglerighteq_{\mathtt{acc}} F\langle x\rangle$ for any $\succeq^{\mathcal{S}}$: $\lambda x.\mathtt{sin}\ F\langle x\rangle \trianglerighteq_{\mathtt{acc}} F\langle x\rangle$ because $\mathtt{sin}\ F\langle x\rangle \trianglerighteq_{\mathtt{acc}} F\langle x\rangle$ because $1 \in Acc(\mathtt{sin})$.

In fact, *all* first-order AFSMs (where all fully applied sub-meta-terms of the left-hand side of a rule have base type) are AFP via the sort ordering $\succeq^{\mathcal{S}}$ that equates all sorts. Also (with the same sort ordering), an AFSM $(\mathcal{F}, \mathcal{R})$ is AFP if, for all rules $\mathtt{f}\ \ell_1 \cdots \ell_k \Rightarrow r \in \mathcal{R}$ and all $1 \leq i \leq k$, we can write: $\ell_i = \lambda x_1 \dots x_{n_i}.\ell'$ where $n_i \geq 0$ and all fully applied sub-meta-terms of ℓ' have base type.

This covers many practical systems, although for Ex. 8 we need a non-trivial sort ordering. Also, there are AFSMs that cannot be handled with *any* $\succeq^{\mathcal{S}}$.

Example 20 (Encoding the untyped λ-calculus). Consider an AFSM with $\mathcal{F} \supseteq \{\mathtt{ap} : \mathtt{o} \rightarrow \mathtt{o} \rightarrow \mathtt{o},\ \mathtt{lm} : (\mathtt{o} \rightarrow \mathtt{o}) \rightarrow \mathtt{o}\}$ and $\mathcal{R} = \{\mathtt{ap}\ (\mathtt{lm}\ F) \Rightarrow F\}$ (note that the only rule has type $\mathtt{o} \rightarrow \mathtt{o}$). This AFSM is not accessible function passing, because $\mathtt{lm}\ F \trianglerighteq_{\mathtt{acc}} F$ cannot hold for any $\succeq^{\mathcal{S}}$ (as this would require $\mathtt{o} \succ^{\mathcal{S}} \mathtt{o}$).

Note that this example is also not terminating. With $t = \mathtt{lm}\ (\lambda x.\mathtt{ap}\ x\ x)$, we get this self-loop as evidence: $\mathtt{ap}\ t\ t \Rightarrow_{\mathcal{R}} (\lambda x.\mathtt{ap}\ x\ x)\ t \Rightarrow_{\beta} \mathtt{ap}\ t\ t$.

Intuitively: in an accessible function passing AFSM, meta-variables of a higher type may occur only in "safe" places in the left-hand sides of rules. Rules like the ones in Ex. 20, where a higher-order meta-variable is lifted out of a base-type term, are not admitted (unless the base type is greater than the higher type).

In the remainder of this paper, we will refer to a *properly applied, accessible function passing* AFSM as a PA-AFP AFSM.

Discussion: This definition is strictly more liberal than the notions of "plain function passing" in both [34] and [46] as adapted to AFSMs. The notion in [46] largely corresponds to AFP if $\succeq^{\mathcal{S}}$ equates all sorts, and the HRS formalism guarantees that rules are properly applied (in fact, all fully applied sub-meta-terms of both left- and right-hand sides of rules have base type). The notion in [34] is more restrictive. The current restriction of PA-AFP AFSMs lets us handle examples like ordinal recursion (Ex. 8) which are not covered by [34,46]. However, note that [34,46] consider a different formalism, which does take rules whose left-hand side is not a pattern into account (which we do not consider). Our restriction also quite resembles the "admissible" rules in [6] which are defined using a pattern computability closure [5], but that work carries additional restrictions.

In later work [32,33], K. Kusakari extends the static DP approach to forms of polymorphic functional programming, with a very liberal restriction: the definition is parametrised with an *arbitrary* RC-set and corresponding accessibility ("safety") notion. Our AFP restriction is actually an instance of this condition (although a more liberal one than the example RC-set used in [32,33]). We have chosen a specific instance because it allows us to use dedicated techniques for the RC-set; for example, our *computable subterm criterion processor* (Thm. 63).

4 Static higher-order dependency pairs

To obtain sufficient criteria for both termination and non-termination of AFSMs, we will now transpose the definition of static dependency pairs [6,34,46,33] to AFSMs. In addition, we will add the new features of *meta-variable conditions*, *formative reductions*, and *computable chains*. Complete versions of all proof sketches in this section are available in Appendix B.

Although we retain the first-order terminology of dependency *pairs*, the setting with meta-variables makes it more suitable to define DPs as *triples*.

Definition 21 ((Static) Dependency Pair). *A* dependency pair (DP) *is a triple $\ell \Rrightarrow p\ (A)$, where ℓ is a closed pattern $\mathtt{f}\ \ell_1 \cdots \ell_k$, p is a closed meta-term $\mathtt{g}\ p_1 \cdots p_n$, and A is a set of* meta-variable conditions*: pairs $Z : i$ indicating that Z regards its i^{th} argument. A DP is* conservative *if $FMV(p) \subseteq FMV(\ell)$.*

A substitution γ respects *a set of meta-variable conditions A if for all $Z : i$ in A we have $\gamma(Z) = \lambda x_1 \dots x_j.t$ with either $i > j$, or $i \leq j$ and $x_i \in FV(t)$. DPs will be used only with substitutions that respect their meta-variable conditions.*

For $\ell \Rrightarrow p\ (\emptyset)$ (so a DP whose set of meta-variable conditions is empty), we often omit the third component and just write $\ell \Rrightarrow p$.

Like the first-order setting, the static DP approach employs *marked function symbols* to obtain meta-terms whose instances cannot be reduced at the root.

Definition 22 (Marked symbols). *Let $(\mathcal{F}, \mathcal{R})$ be an AFSM. Define $\mathcal{F}^\sharp := \mathcal{F} \uplus \{\mathtt{f}^\sharp : \sigma \mid \mathtt{f} : \sigma \in \mathcal{D}\}$. For a meta-term $s = \mathtt{f}\ s_1 \cdots s_k$ with $\mathtt{f} \in \mathcal{D}$ and $k = minar(\mathtt{f})$, we let $s^\sharp = \mathtt{f}^\sharp\ s_1 \cdots s_k$; for s of other forms $s^\sharp$ is not defined.*

Moreover, we will consider *candidates*. In the first-order setting, candidate terms are subterms of the right-hand sides of rules whose root symbol is a defined symbol. Intuitively, these subterms correspond to function calls. In the current setting, we have to consider also meta-variables as well as rules whose right-hand side is not β-normal (which might arise for instance due to η-expansion).

Definition 23 (β-reduced-sub-meta-term, $\trianglerighteq_\beta$, $\trianglerighteq_A$). *A meta-term s has a fully applied* β-reduced-sub-meta-term *t (shortly,* BRSMT*), notation $s \trianglerighteq_\beta t$, if there exists a set of meta-variable conditions A with $s \trianglerighteq_A t$. Here $s \trianglerighteq_A t$ holds if:*

- *$s = t$, or*
- *$s = \lambda x.u$ and $u \trianglerighteq_A t$, or*
- *$s = (\lambda x.u)\ s_0 \cdots s_n$ and some $s_i \trianglerighteq_A t$, or $u[x := s_0]\ s_1 \cdots s_n \trianglerighteq_A t$, or*
- *$s = a\ s_1 \cdots s_n$ with $a \in \mathcal{F} \cup \mathcal{V}$ and some $s_i \trianglerighteq_A t$, or*
- *$s = Z\langle t_1, \dots, t_k\rangle\ s_1 \cdots s_n$ and some $s_i \trianglerighteq_A t$, or*
- *$s = Z\langle t_1, \dots, t_k\rangle\ s_1 \cdots s_n$ and $t_i \trianglerighteq_A t$ for some $i \in \{1, \dots, k\}$ with $(Z : i) \in A$.*

Essentially, $s \trianglerighteq_A t$ means that t can be reached from s by taking β-reductions at the root and "subterm"-steps, where $Z : i$ is in A whenever we pass into argument i of a meta-variable Z. BRSMTs are used to generate *candidates*:

Definition 24 (Candidates). *For a meta-term s, the set $\mathsf{cand}(s)$ of* candidates of s *consists of those pairs $t\ (A)$ such that (a) t has the form $\mathtt{f}\ s_1 \cdots s_k$ with $\mathtt{f} \in \mathcal{D}$ and $k = minar(\mathtt{f})$, and (b) there are $s_{k+1}, \dots, s_n$ (with $n \geq k$) such that $s \trianglerighteq_A t\ s_{k+1} \cdots s_n$, and (c) A is minimal: there is no subset $A' \subsetneq A$ with $s \trianglerighteq_{A'} t$.*

Example 25. In AFSMs where all meta-variables have arity 0 and the right-hand sides of rules are β-normal, the set $\mathsf{cand}(s)$ for a meta-term s consists exactly of the pairs $t\ (\emptyset)$ where t has the form $\mathtt{f}\ s_1 \cdots s_{minar(\mathtt{f})}$ and t occurs as part of s. In Ex. 8, we thus have $\mathsf{cand}(G\ H\ (\lambda m.\mathtt{rec}\ (H\ m)\ K\ F\ G)) = \{\, \mathtt{rec}\ (H\ m)\ K\ F\ G\ (\emptyset)\,\}$.

If some of the meta-variables *do* take arguments, then the meta-variable conditions matter: candidates of s are pairs $t\ (A)$ where A contains exactly those pairs $Z : i$ for which we pass through the i^{th} argument of Z to reach t in s.

Example 26. Consider an AFSM with the signature from Ex. 8 but a rule using meta-variables with larger arities:

$$\begin{gathered}\mathtt{rec}\ (\mathtt{lim}\ (\lambda n.H\langle n\rangle))\ K\ (\lambda x.\lambda n.F\langle x,n\rangle)\ (\lambda f.\lambda g.G\langle f,g\rangle)\ \Rightarrow \\ G\langle \lambda n.H\langle n\rangle,\ \lambda m.\mathtt{rec}\ H\langle m\rangle\ K\ (\lambda x.\lambda n.F\langle x,n\rangle)\ (\lambda f.\lambda g.G\langle f,g\rangle)\rangle\end{gathered}$$

The right-hand side has one candidate:

$$\mathtt{rec}\ H\langle m\rangle\ K\ (\lambda x.\lambda n.F\langle x,n\rangle)\ (\lambda f.\lambda g.G\langle f,g\rangle)\ (\{G : 2\})$$

The original static approaches define DPs as pairs $\ell^\sharp \Rrightarrow p^\sharp$ where $\ell \Rightarrow r$ is a rule and p a subterm of r of the form $\mathtt{f}\ r_1 \cdots r_m$ – as their rules are built using terms, not meta-terms. This can set variables bound in r free in p. In the current setting, we use candidates with their meta-variable conditions and implicit β-steps rather than subterms, and we replace such variables by meta-variables.

Definition 27 (*SDP*). *Let s be a meta-term and $(\mathcal{F}, \mathcal{R})$ be an AFSM. Let $metafy(s)$ denote s with all free variables replaced by corresponding meta-variables. Now $SDP(\mathcal{R}) = \{\ell^\sharp \Rrightarrow metafy(p^\sharp)\ (A) \mid \ell \Rightarrow r \in \mathcal{R} \wedge p\ (A) \in \mathsf{cand}(r)\}$.*

Although static DPs always have a pleasant form $\mathtt{f}^\sharp\ \ell_1 \cdots \ell_k \Rrightarrow \mathtt{g}^\sharp\ p_1 \cdots p_n\ (A)$ (as opposed to the *dynamic* DPs of, e.g., [31], whose right-hand sides can have a meta-variable at the head, which complicates various techniques in the framework), they have two important complications not present in first-order DPs: the right-hand side p of a DP $\ell \Rrightarrow p\ (A)$ may contain meta-variables that do not occur in the left-hand side ℓ – traditional analysis techniques are not really equipped for this – and the left- and right-hand sides may have different types. In § 5 we will explore some methods to deal with these features.

Example 28. For the non-η-expanded rules of Ex. 17, the set $SDP(\mathcal{R})$ has one element: $\mathtt{deriv}^\sharp\ (\lambda x.\mathtt{sin}\ F\langle x\rangle) \Rrightarrow \mathtt{deriv}^\sharp\ (\lambda x.F\langle x\rangle)$. (As `times` and `cos` are not defined symbols, they do not generate dependency pairs.) The set $SDP(\mathcal{R}^\uparrow)$ for the η-expanded rules is $\{\mathtt{deriv}^\sharp\ (\lambda x.\mathtt{sin}\ F\langle x\rangle)\ Y \Rrightarrow \mathtt{deriv}^\sharp\ (\lambda x.F\langle x\rangle)\ Y\}$. To obtain the relevant candidate, we used the β-reduction step of BRSMTs.

Example 29. The AFSM from Ex. 8 is AFP following Ex. 19; here $SDP(\mathcal{R})$ is:

$$\mathtt{rec}^\sharp\ (\mathtt{s}\ X)\ K\ F\ G \Rrightarrow \mathtt{rec}^\sharp\ X\ K\ F\ G\ (\emptyset)$$
$$\mathtt{rec}^\sharp\ (\mathtt{lim}\ H)\ K\ F\ G \Rrightarrow \mathtt{rec}^\sharp\ (H\ M)\ K\ F\ G\ (\emptyset)$$

Note that the right-hand side of the second DP contains a meta-variable that is not on the left. As we will see in Ex. 64, that is not problematic here.

Termination analysis using dependency pairs importantly considers the notion of a *dependency chain*. This notion is fairly similar to the first-order setting:

Definition 30 (Dependency chain). *Let $\mathcal{P}$ be a set of DPs and $\mathcal{R}$ a set of rules. A (finite or infinite) $(\mathcal{P}, \mathcal{R})$-*dependency chain* (or just $(\mathcal{P}, \mathcal{R})$-chain) is a sequence $[(\ell_0 \Rrightarrow p_0\ (A_0), s_0, t_0), (\ell_1 \Rrightarrow p_1\ (A_1), s_1, t_1), \dots]$ where each $\ell_i \Rrightarrow p_i\ (A_i) \in \mathcal{P}$ and all s_i, t_i are terms, such that for all i:*

1. *there exists a substitution γ on domain $FMV(\ell_i) \cup FMV(p_i)$ such that $s_i = \ell_i\gamma,\ t_i = p_i\gamma$ and for all $Z \in \mathtt{dom}(\gamma)$: $\gamma(Z)$ respects A_i;*
2. *we can write $t_i = \mathtt{f}\ u_1 \cdots u_n$ and $s_{i+1} = \mathtt{f}\ w_1 \cdots w_n$ and each $u_j \Rightarrow^*_{\mathcal{R}} w_j$.*

Example 31. In the (first) AFSM from Ex. 6, we have $SDP(\mathcal{R}) = \{\mathtt{map}^\sharp\ (\lambda x.Z\langle x\rangle)\ (\mathtt{cons}\ H\ T) \Rrightarrow \mathtt{map}^\sharp\ (\lambda x.Z\langle x\rangle)\ T\}$. An example of a finite dependency chain is $[(\rho, s_1, t_1), (\rho, s_2, t_2)]$ where ρ is the one DP, $s_1 = \mathtt{map}^\sharp\ (\lambda x.\mathtt{s}\ x)\ (\mathtt{cons}\ \mathtt{0}\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ (\mathtt{map}\ (\lambda x.x)\ \mathtt{nil})))$ and $t_1 = \mathtt{map}^\sharp\ (\lambda x.\mathtt{s}\ x)\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ (\mathtt{map}\ (\lambda x.x)\ \mathtt{nil}))$ and $s_2 = \mathtt{map}^\sharp\ (\lambda x.\mathtt{s}\ x)\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{nil})$ and $t_2 = \mathtt{map}^\sharp\ (\lambda x.\mathtt{s}\ x)\ \mathtt{nil}$.

Note that here t_1 reduces to s_2 in a single step ($\mathtt{map}\ (\lambda x.x)\ \mathtt{nil} \Rightarrow_{\mathcal{R}} \mathtt{nil}$).

We have the following key result:

Theorem 32. *Let $(\mathcal{F}, \mathcal{R})$ be a PA-AFP AFSM. If $(\mathcal{F}, \mathcal{R})$ is non-terminating, then there is an infinite $(SDP(\mathcal{R}), \mathcal{R})$-dependency chain.*

Proof (sketch). The proof is an adaptation of the one in [34], altered for the more permissive definition of *accessible function passing* over *plain function passing* as well as the meta-variable conditions; it also follows from Thm. 37 below. □

By this result we can use dependency pairs to prove termination of a given properly applied and AFP AFSM: if we can prove that there is no infinite $(\mathit{SDP}(\mathcal{R}),\mathcal{R})$-chain, then termination follows immediately. Note, however, that the reverse result does *not* hold: it is possible to have an infinite $(\mathit{SDP}(\mathcal{R}),\mathcal{R})$-dependency chain even for a terminating PA-AFP AFSM.

Example 33. Let $\mathcal{F} \supseteq \{\mathtt{0},\mathtt{1} : \mathtt{nat},\ \mathtt{f} : \mathtt{nat} \to \mathtt{nat},\ \mathtt{g} : (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}\}$ and $\mathcal{R} = \{\mathtt{f}\ \mathtt{0} \Rightarrow \mathtt{g}\ (\lambda x.\mathtt{f}\ x), \mathtt{g}\ (\lambda x.F\langle x\rangle) \Rightarrow F\langle \mathtt{1}\rangle\}$. This AFSM is PA-AFP, with $\mathit{SDP}(\mathcal{R}) = \{\mathtt{f}^\sharp\ \mathtt{0} \Rrightarrow \mathtt{g}^\sharp\ (\lambda x.\mathtt{f}\ x),\ \mathtt{f}^\sharp\ \mathtt{0} \Rrightarrow \mathtt{f}^\sharp\ X\}$; the second rule does not cause the addition of any dependency pairs. Although $\Rightarrow_{\mathcal{R}}$ is terminating, there is an infinite $(\mathit{SDP}(\mathcal{R}),\mathcal{R})$-chain $[(\mathtt{f}^\sharp\ \mathtt{0} \Rrightarrow \mathtt{f}^\sharp\ X, \mathtt{f}^\sharp\ \mathtt{0}, \mathtt{f}^\sharp\ \mathtt{0}), (\mathtt{f}^\sharp\ \mathtt{0} \Rrightarrow \mathtt{f}^\sharp\ X, \mathtt{f}^\sharp\ \mathtt{0}, \mathtt{f}^\sharp\ \mathtt{0}), \ldots]$.

The problem in Ex. 33 is the *non-conservative* DP $\mathtt{f}^\sharp\ \mathtt{0} \Rrightarrow \mathtt{f}^\sharp\ X$, with X on the right but not on the left. Such DPs arise from *abstractions* in the right-hand sides of rules. Unfortunately, abstractions are introduced by the restricted η-expansion (Def. 15) that we may need to make an AFSM properly applied. Even so, often all DPs are conservative, like Ex. 6 and 17. There, we do have the inverse result:

Theorem 34. *For any AFSM $(\mathcal{F},\mathcal{R})$: if there is an infinite $(\mathit{SDP}(\mathcal{R}),\mathcal{R})$-chain $[(\rho_0,s_0,t_0),(\rho_1,s_1,t_1),\ldots]$ with all ρ_i conservative, then $\Rightarrow_{\mathcal{R}}$ is non-terminating.*

Proof (sketch). If $\mathit{FMV}(p_i) \subseteq \mathit{FMV}(\ell_i)$, then we can see that $s_i \Rightarrow_{\mathcal{R}} \cdot \Rightarrow_\beta^* t_i'$ for some term t_i' of which t_i is a subterm. Since also each $t_i \Rightarrow_{\mathcal{R}}^* s_{i+1}$, the infinite chain induces an infinite reduction $s_0 \Rightarrow_{\mathcal{R}}^+ t_0' \Rightarrow_{\mathcal{R}}^* s_1' \Rightarrow_{\mathcal{R}}^+ t_1'' \Rightarrow_{\mathcal{R}}^* \ldots$. □

The core of the dependency pair *framework* is to systematically simplify a set of pairs $(\mathcal{P},\mathcal{R})$ to prove either absence or presence of an infinite $(\mathcal{P},\mathcal{R})$-chain, thus showing termination or non-termination as appropriate. By Theorems 32 and 34 we can do so, although with some conditions on the non-termination result. We can do better by tracking certain properties of dependency chains.

Definition 35 (Minimal and Computable chains). *Let $(\mathcal{F},\mathcal{U})$ be an AFSM and $C_{\mathcal{U}}$ an RC-set satisfying the properties of Thm. 13 for $(\mathcal{F},\mathcal{U})$. Let $\mathcal{F}$ contain, for every type σ, at least countably many symbols $\mathtt{f} : \sigma$ not used in $\mathcal{U}$.*

A $(\mathcal{P},\mathcal{R})$-chain $[(\rho_0,s_0,t_0),(\rho_1,s_1,t_1),\ldots]$ is $\mathcal{U}$-computable *if: $\Rightarrow_{\mathcal{U}} \supseteq \Rightarrow_{\mathcal{R}}$, and for all $i \in \mathbb{N}$ there exists a substitution γ_i such that $\rho_i = \ell_i \Rrightarrow p_i\ (A_i)$ with $s_i = \ell_i\gamma_i$ and $t_i = p_i\gamma_i$, and $(\lambda x_1 \ldots x_n.v)\gamma_i$ is $C_{\mathcal{U}}$-computable for all v and B such that $p_i \trianglerighteq_B v$, γ_i respects B, and $FV(v) = \{x_1,\ldots,x_n\}$.*

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

In the first-order DP framework, *minimal* chains give access to several powerful techniques to prove absence of infinite chains, such as the *subterm criterion* [24] and *usable rules* [22,24]. *Computable* chains go a step further, by building on the computability inherent in the proof of Thm. 32 and the notion of *accessible* *function passing* AFSMs. In computable chains, we can require that (some of) the subterms of all t_i are *computable* rather than merely *terminating*. This property will be essential in the *computable subterm criterion processor* (Thm. 63).

Another property of dependency chains is the use of *formative rules*, which has proven very useful for dynamic DPs [31]. Here we go further and consider *formative reductions*, which were introduced for the first-order DP framework in [16]. This property will be essential in the *formative rules processor* (Thm. 58).

Definition 36 (Formative chain, formative reduction). *A $(\mathcal{P}, \mathcal{R})$-chain $[(\ell_0 \Rrightarrow p_0\ (A_0), s_0, t_0), (\ell_1 \Rrightarrow p_1\ (A_1), s_1, t_1), \ldots]$ is* formative *if for all i, the reduction $t_i \Rightarrow^*_{\mathcal{R}} s_{i+1}$ is ℓ_{i+1}-formative. Here, for a pattern ℓ, substitution γ and term s, a reduction $s \Rightarrow^*_{\mathcal{R}} \ell\gamma$ is ℓ-*formative *if one of the following holds:*

- *ℓ is not a fully extended linear pattern; that is: some meta-variable occurs more than once in ℓ or ℓ has a sub-meta-term $\lambda x.C[Z\langle \boldsymbol{s} \rangle]$ with $x \notin \{\boldsymbol{s}\}$*
- *ℓ is a meta-variable application $Z\langle x_1, \ldots, x_k \rangle$ and $s = \ell\gamma$*
- *$s = a\ s_1 \cdots s_n$ and $\ell = a\ \ell_1 \cdots \ell_n$ with $a \in \mathcal{F}^\sharp \cup \mathcal{V}$ and each $s_i \Rightarrow^*_{\mathcal{R}} \ell_i\gamma$ by an ℓ_i-formative reduction*
- *$s = \lambda x.s'$ and $\ell = \lambda x.\ell'$ and $s' \Rightarrow^*_{\mathcal{R}} \ell'\gamma$ by an ℓ'-formative reduction*
- *$s = (\lambda x.u)\ v\ w_1 \cdots w_n$ and $u[x := v]\ w_1 \cdots w_n \Rightarrow^*_{\mathcal{R}} \ell\gamma$ by an ℓ-formative reduction*
- *ℓ is not a meta-variable application, and there are $\ell' \Rightarrow r' \in \mathcal{R}$, meta-variables $Z_1 \ldots Z_n$ $(n \geq 0)$ and δ such that $s \Rightarrow^*_{\mathcal{R}} (\ell'\ Z_1 \cdots Z_n)\delta$ by an $(\ell'\ Z_1 \cdots Z_n)$-formative reduction, and $(r'\ Z_1 \cdots Z_n)\delta \Rightarrow^*_{\mathcal{R}} \ell\gamma$ by an ℓ-formative reduction.*

The idea of a formative reduction is to avoid redundant steps: if $s \Rightarrow^*_{\mathcal{R}} \ell\gamma$ by an ℓ-formative reduction, then this reduction takes only the steps needed to obtain an instance of ℓ. Suppose that we have rules $\mathtt{plus}\ \mathtt{0}\ Y \Rightarrow Y$, $\mathtt{plus}\ (\mathtt{s}\ X)\ Y \Rightarrow \mathtt{s}\ (\mathtt{plus}\ X\ Y)$. Let $\ell := \mathtt{g}\ \mathtt{0}\ X$ and $t := \mathtt{plus}\ \mathtt{0}\ \mathtt{0}$. Then the reduction $\mathtt{g}\ t\ t \Rightarrow_{\mathcal{R}} \mathtt{g}\ \mathtt{0}\ t$ is ℓ-formative: we must reduce the first argument to get an instance of ℓ. The reduction $\mathtt{g}\ t\ t \Rightarrow_{\mathcal{R}} \mathtt{g}\ t\ \mathtt{0} \Rightarrow_{\mathcal{R}} \mathtt{g}\ \mathtt{0}\ \mathtt{0}$ is not ℓ-formative, because the reduction in the second argument does not contribute to the non-meta-variable positions of ℓ. This matters when we consider ℓ as the left-hand side of a rule, say $\mathtt{g}\ \mathtt{0}\ X \Rightarrow \mathtt{0}$: if we reduce $\mathtt{g}\ t\ t \Rightarrow_{\mathcal{R}} \mathtt{g}\ t\ \mathtt{0} \Rightarrow_{\mathcal{R}} \mathtt{g}\ \mathtt{0}\ \mathtt{0} \Rightarrow_{\mathcal{R}} \mathtt{0}$, then the first step was redundant: removing this step gives a shorter reduction to the same result: $\mathtt{g}\ t\ t \Rightarrow_{\mathcal{R}} \mathtt{g}\ \mathtt{0}\ t \Rightarrow_{\mathcal{R}} \mathtt{0}$. In an infinite reduction, redundant steps may also be postponed indefinitely.

We can now strengthen the result of Thm. 32 with two new properties.

Theorem 37. *Let $(\mathcal{F}, \mathcal{R})$ be a* properly applied, *accessible function passing AFSM. If $(\mathcal{F}, \mathcal{R})$ is non-terminating, then there is an infinite $\mathcal{R}$-computable formative $(SDP(\mathcal{R}), \mathcal{R})$-dependency chain.*

Proof (sketch). We select a *minimal non-computable (MNC)* term $s := \mathtt{f}\ s_1 \cdots s_k$ (where all s_i are $C_{\mathcal{R}}$-computable) and an infinite reduction starting in s. Then we stepwise build an infinite dependency chain, as follows. Since s is non-computable but each s_i terminates (as computability implies termination), there exist a rule $\mathtt{f}\ \ell_1 \cdots \ell_k \Rightarrow r$ and substitution γ such that each $s_i \Rightarrow_{\mathcal{R}}^* \ell_i\gamma$ and $r\gamma$ is non-computable. We can then identify a candidate t (A) of r such that γ respects A and $t\gamma$ is a MNC subterm of $r\gamma$; we continue the process with $t\gamma$ (or a term at its head). For the *formative* property, we note that if $s \Rightarrow_{\mathcal{R}}^* \ell\gamma$ and u is terminating, then $u \Rightarrow_{\mathcal{R}}^* \ell\delta$ by an ℓ-formative reduction for substitution δ such that each $\delta(Z) \Rightarrow_{\mathcal{R}}^* \gamma(Z)$. This follows by postponing those reduction steps not needed to obtain an instance of ℓ. The resulting infinite chain is $\mathcal{R}$-computable because we can show, by induction on the definition of $\trianglerighteq_{\mathtt{acc}}$, that if $\ell \Rightarrow r$ is an AFP rule and $\ell\gamma$ is a MNC term, then $\gamma(Z)$ is $C_{\mathcal{R}}$-computable for all $Z \in \mathit{FMV}(r)$. $\square$

As it is easily seen that all $C_{\mathcal{U}}$-computable terms are $\Rightarrow_{\mathcal{U}}$-terminating and therefore $\Rightarrow_{\mathcal{R}}$-terminating, every $\mathcal{U}$-computable $(\mathcal{P}, \mathcal{R})$-dependency chain is also minimal. The notions of $\mathcal{R}$-computable and formative chains still do not suffice to obtain a true inverse result, however (i.e., to prove that termination implies the absence of an infinite $\mathcal{R}$-computable chain over $\mathit{SDP}(\mathcal{R})$): the infinite chain in Ex. 33 is $\mathcal{R}$-computable.

To see why the two restrictions that the AFSM must be *properly applied* and *accessible function passing* are necessary, consider the following examples.

Example 38. Consider $\mathcal{F} \supseteq \{\mathtt{fix} : ((\mathtt{o} \rightarrow \mathtt{o}) \rightarrow \mathtt{o} \rightarrow \mathtt{o}) \rightarrow \mathtt{o} \rightarrow \mathtt{o}\}$ and $\mathcal{R} = \{\mathtt{fix}\ F\ X \Rightarrow F\ (\mathtt{fix}\ F)\ X\}$. This AFSM is not properly applied; it is also not terminating, as can be seen by instantiating F with $\lambda y.y$. However, it does not have any static DPs, since $\mathtt{fix}\ F$ is not a candidate. Even if we altered the definition of static DPs to admit a dependency pair $\mathtt{fix}^\sharp\ F\ X \Rrightarrow \mathtt{fix}^\sharp\ F$, this pair could not be used to build an infinite dependency chain.

Note that the problem does not arise if we study the η-expanded rules $\mathcal{R}^\uparrow = \{\mathtt{fix}\ F\ X \Rightarrow F\ (\lambda z.\mathtt{fix}\ F\ z)\ X\}$, as the dependency pair $\mathtt{fix}^\sharp\ F\ X \Rrightarrow \mathtt{fix}^\sharp\ F\ Z$ does admit an infinite chain. Unfortunately, as the one dependency pair does not satisfy the conditions of Thm. 34, we cannot use this to prove non-termination.

Example 39. The AFSM from Ex. 20 is not accessible function passing, since $\mathit{Acc}(\mathtt{lm}) = \emptyset$. This is good because the set $\mathit{SDP}(\mathcal{R})$ is empty, which would lead us to falsely conclude termination without the restriction.

Discussion: Thm. 37 transposes the work of [34,46] to AFSMs and extends it by using a more liberal restriction, by limiting interest to *formative*, *$\mathcal{R}$-computable* chains, and by including meta-variable conditions. Both of these new properties of chains will support new termination techniques within the DP framework.

The relationship with the works for functional programming [32,33] is less clear: they define a different form of chains suited well to polymorphic systems, but which requires more intricate reasoning for non-polymorphic systems, as DPs can be used for reductions at the head of a term. It is not clear whether there are non-polymorphic systems that can be handled with one and not the other. The notions of formative and $\mathcal{R}$-computable chains are not considered there; meta-variable conditions are not relevant to their λ-free formalism.

5 The static higher-order DP framework

In first-order term rewriting, the DP *framework* [20] is an extendable framework to prove termination and non-termination. As observed in the introduction, DP analyses in higher-order rewriting typically go beyond the initial DP *approach* [2], but fall short of the full *framework*. Here, we define the latter for static DPs. Complete versions of all proof sketches in this section are available in Appendix C.

We have now reduced the problem of termination to non-existence of certain chains. In the DP framework, we formalise this in the notion of a *DP problem*:

Definition 40 (DP problem). *A* DP problem *is a tuple* $(\mathcal{P}, \mathcal{R}, m, f)$ *with* $\mathcal{P}$ *a set of DPs,* $\mathcal{R}$ *a set of rules,* $m \in \{\mathtt{minimal}, \mathtt{arbitrary}\} \cup \{\mathtt{computable}_{\mathcal{U}} \mid$ *any set of rules* $\mathcal{U}\}$*, and* $f \in \{\mathtt{formative}, \mathtt{all}\}$.[5]

A DP problem $(\mathcal{P}, \mathcal{R}, m, f)$ *is* finite *if there exists no infinite* $(\mathcal{P}, \mathcal{R})$*-chain that is* $\mathcal{U}$*-computable if* $m = \mathtt{computable}_{\mathcal{U}}$*, is minimal if* $m = \mathtt{minimal}$*, and is formative if* $f = \mathtt{formative}$*. It is* infinite *if* $\mathcal{R}$ *is non-terminating, or if there exists an infinite* $(\mathcal{P}, \mathcal{R})$*-chain where all DPs used in the chain are conservative.*

To capture the levels of permissiveness in the m flag, we use a transitive-reflexive relation $\succeq$ *generated by* $\mathtt{computable}_{\mathcal{U}} \succeq \mathtt{minimal} \succeq \mathtt{arbitrary}$*.*

Thus, the combination of Theorems 37 and 34 can be rephrased as: an AFSM $(\mathcal{F}, \mathcal{R})$ is terminating if $(\mathit{SDP}(\mathcal{R}), \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative})$ is finite, and is non-terminating if $(\mathit{SDP}(\mathcal{R}), \mathcal{R}, m, f)$ is infinite for some $m \in \{\mathtt{computable}_{\mathcal{U}}, \mathtt{minimal}, \mathtt{arbitrary}\}$ and $f \in \{\mathtt{formative}, \mathtt{all}\}$.[6]

The core idea of the DP framework is to iteratively simplify a set of DP problems via *processors* until nothing remains to be proved:

Definition 41 (Processor). *A* dependency pair processor *(or just* processor*) is a function that takes a DP problem and returns either* NO *or a set of DP problems. A processor Proc is* sound *if a DP problem M is finite whenever* $\mathit{Proc}(M) \neq$ NO *and all elements of* $\mathit{Proc}(M)$ *are finite. A processor Proc is* complete *if a DP problem M is infinite whenever* $\mathit{Proc}(M) =$ NO *or contains an infinite element.*

To prove finiteness of a DP problem M with the DP framework, we proceed analogously to the first-order DP framework [22]: we repeatedly apply sound DP processors starting from M until none remain. That is, we execute the following rough procedure: (1) let $A := \{M\}$; (2) while $A \neq \emptyset$: select a problem $Q \in A$ and a sound processor *Proc* with $\mathit{Proc}(Q) \neq \mathtt{NO}$, and let $A := (A \setminus \{Q\}) \cup \mathit{Proc}(Q)$. If this procedure terminates, then M is a finite DP problem.

[5] Our framework is implicitly parametrised by the signature $\mathcal{F}^\sharp$ used for term formation. As none of the processors we present modify this component (as indeed there is no need to by Thm. 9), we leave it implicit.

[6] The processors in this paper do not *alter* the flag m, but some *require* minimality or computability. We include the `minimal` option and the subscript $\mathcal{U}$ for the sake of future generalisations, and for reuse of processors in the *dynamic* approach of [31].

To prove termination of an AFSM $(\mathcal{F}, \mathcal{R})$, we would use as initial DP problem $(\mathit{SDP}(\mathcal{R}), \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative})$, provided that $\mathcal{R}$ is properly applied and accessible function passing (where η-expansion following Def. 15 may be applied first). If the procedure terminates – so finiteness of M is proved by the definition of soundness – then Thm. 37 provides termination of $\Rightarrow_{\mathcal{R}}$.

Similarly, we can use the DP framework to prove infiniteness: (1) let $A := \{M\}$; (2) while $A \neq \mathtt{NO}$: select a problem $Q \in A$ and a complete processor *Proc*, and let $A := \mathtt{NO}$ if $\mathit{Proc}(Q) = \mathtt{NO}$, or $A := (A \setminus \{Q\}) \cup \mathit{Proc}(Q)$ otherwise. For non-termination of $(\mathcal{F}, \mathcal{R})$, the initial DP problem should be $(\mathit{SDP}(\mathcal{R}), \mathcal{R}, m, f)$, where m, f can be any flag (see Thm. 34). Note that the algorithms coincide while processors are used that are both sound *and* complete. In a tool, automation (or the user) must resolve the non-determinism and select suitable processors.

Below, we will present a number of processors within the framework. We will typically present processors by writing "for a DP problem M satisfying X, Y, Z, $\mathit{Proc}(M) = \dots$". In these cases, we let $\mathit{Proc}(M) = \{M\}$ for any problem M not satisfying the given properties. Many more processors are possible, but we have chosen to present a selection which touches on all aspects of the DP framework:

– processors which map a DP problem to $\mathtt{NO}$ (Thm. 65), a singleton set (most processors) and a non-singleton set (Thm. 42);
– changing the set $\mathcal{R}$ (Thm. 58, 54) and various flags (Thm. 54);
– using specific values of the f (Thm. 58) and m flags (Thm. 61, 54, 63);
– using term orderings (Thm. 49, 52), a key part of many termination proofs.

5.1 The dependency graph

We can leverage reachability information to *decompose* DP problems. In first-order rewriting, a graph structure is used to track which DPs can possibly follow one another in a chain [2]. Here, we define this *dependency graph* as follows.

Definition 42 (Dependency graph). *A DP problem $(\mathcal{P}, \mathcal{R}, m, f)$ induces a graph structure DG, called its* dependency graph*, whose nodes are the elements of $\mathcal{P}$. There is a (directed) edge from ρ_1 to ρ_2 in DG iff there exist s_1, t_1, s_2, t_2 such that $[(\rho_1, s_1, t_1), (\rho_2, s_2, t_2)]$ is a $(\mathcal{P}, \mathcal{R})$-chain with the properties for m, f.*

Example 43. Consider an AFSM with $\mathcal{F} \supseteq \{\mathtt{f} : (\mathtt{nat} \rightarrow \mathtt{nat}) \rightarrow \mathtt{nat} \rightarrow \mathtt{nat}\}$ and $\mathcal{R} = \{\mathtt{f}\ (\lambda x.F\langle x\rangle)\ (\mathtt{s}\ Y) \Rightarrow F\langle \mathtt{f}\ (\lambda x.\mathtt{0})\ (\mathtt{f}\ (\lambda x.F\langle x\rangle)\ Y)\rangle\}$. Let $\mathcal{P} := \mathit{SDP}(\mathcal{R}) =$

$$\left\{ \begin{array}{ll} (1)\ \mathtt{f}^\sharp\ (\lambda x.F\langle x\rangle)\ (\mathtt{s}\ Y) \Rrightarrow \mathtt{f}^\sharp\ (\lambda x.\mathtt{0})\ (\mathtt{f}\ (\lambda x.F\langle x\rangle)\ Y) & (\{F : 1\}) \\ (2)\ \mathtt{f}^\sharp\ (\lambda x.F\langle x\rangle)\ (\mathtt{s}\ Y) \Rrightarrow \mathtt{f}^\sharp\ (\lambda x.F\langle x\rangle)\ Y & (\{F : 1\}) \end{array} \right\}$$

The dependency graph of $(\mathcal{P}, \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$ is:

$$\begin{tikzcd} \boxed{(1)} & \boxed{(2)} \arrow[l] \arrow[loop right] \end{tikzcd}$$

There is no edge from (1) to itself or (2) because there is no substitution γ such that $(\lambda x.\mathtt{0})\gamma$ can be reduced to a term $(\lambda x.F\langle x\rangle)\delta$ where $\delta(F)$ regards its first argument (as $\Rightarrow^*_{\mathcal{R}}$ cannot introduce new variables).

In general, the dependency graph for a given DP problem is undecidable, which is why we consider *approximations*.

Definition 44 (Dependency graph approximation [31]). *A finite graph* G_θ approximates *DG if θ is a function that maps the nodes of DG to the nodes of G_θ such that, whenever DG has an edge from ρ_1 to ρ_2, G_θ has an edge from $\theta(\rho_1)$ to $\theta(\rho_2)$. (G_θ may have edges that have no corresponding edge in DG.)*

Note that this definition allows for an *infinite* graph to be approximated by a *finite* one; infinite graphs may occur if $\mathcal{R}$ is infinite (e.g., the union of all simply-typed instances of polymorphic rules).

If $\mathcal{P}$ is finite, we can take a graph approximation $G_{\mathtt{id}}$ with the same nodes as DG. A simple approximation may have an edge from $\ell_1 \Rrightarrow p_1\ (A_1)$ to $\ell_2 \Rrightarrow p_2\ (A_2)$ whenever both p_1 and ℓ_2 have the form $\mathtt{f}^\sharp\ s_1 \cdots s_k$ for the same $\mathtt{f}$ and k. However, one can also take the meta-variable conditions into account, as we did in Ex. 43.

Theorem 45 (Dependency graph processor). *The processor $Proc_{G_\theta}$ that maps a DP problem $M = (\mathcal{P}, \mathcal{R}, m, f)$ to $\{(\{\rho \in \mathcal{P} \mid \theta(\rho) \in C_i\}, \mathcal{R}, m, f) \mid 1 \leq i \leq n\}$ if G_θ is an approximation of the dependency graph of M and $C_1, \dots, C_n$ are the (nodes of the) non-trivial strongly connected components (SCCs) of G_θ, is both sound and complete.*

Proof (sketch). In an infinite $(\mathcal{P}, \mathcal{R})$-chain $[(\rho_0, s_0, t_0), (\rho_1, s_1, t_1), \dots]$, there is always a path from ρ_i to ρ_{i+1} in DG. Since G_θ is finite, every infinite path in DG eventually remains in a cycle in G_θ. This cycle is part of an SCC. □

Example 46. Let $\mathcal{R}$ be the set of rules from Ex. 43 and G be the graph given there. Then $Proc_G(SDP(\mathcal{R}), \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative}) = \{(\{\mathtt{f}^\sharp\ (\lambda x.F\langle x\rangle)\ (\mathtt{s}\ Y) \Rrightarrow \mathtt{f}^\sharp\ (\lambda x.F\langle x\rangle)\ Y\ (\{F : 1\})\}, \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative})\}$.

Example 47. Let $\mathcal{R}$ consist of the rules for $\mathtt{map}$ from Ex. 6 along with $\mathtt{f}\ L \Rightarrow \mathtt{map}\ (\lambda x.\mathtt{g}\ x)\ L$ and $\mathtt{g}\ X \Rightarrow X$. Then $SDP(\mathcal{R}) = \{(1)\ \mathtt{map}^\sharp\ (\lambda x.Z\langle x\rangle)\ (\mathtt{cons}\ H\ T) \Rrightarrow \mathtt{map}^\sharp\ (\lambda x.Z\langle x\rangle)\ T,\ (2)\ \mathtt{f}^\sharp\ L \Rrightarrow \mathtt{map}^\sharp\ (\lambda x.\mathtt{g}\ x)\ L,\ (3)\ \mathtt{f}^\sharp\ L \Rrightarrow \mathtt{g}^\sharp\ X\}$. DP (3) is not conservative, but it is not on any cycle in the graph approximation $G_{\mathtt{id}}$ obtained by considering head symbols as described above:

$$\begin{tikzcd}
\boxed{(3)} & \boxed{(2)} \arrow[r] & \boxed{(1)} \arrow[loop above]
\end{tikzcd}$$

As (1) is the only DP on a cycle, $Proc_{SDP_{G_{\mathtt{id}}}}(SDP(\mathcal{R}), \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative}) = \{\ (\{(1)\}, \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative})\ \}$.

Discussion: The dependency graph is a powerful tool for simplifying DP problems, used since early versions of the DP approach [2]. Our notion of a dependency graph approximation, taken from [31], strictly generalises the original notion in [2], which uses a graph on the same node set as DG with possibly further edges. One can get this notion here by using a graph $G_{\mathtt{id}}$. The advantage of our definition is that it ensures soundness of the dependency graph processor also for *infinite* sets of DPs. This overcomes a restriction in the literature [34, Corollary 5.13] to dependency graphs without non-cyclic infinite paths.

5.2 Processors based on reduction triples

At the heart of most DP-based approaches to termination proving lie well-founded orderings to delete DPs (or rules). For this, we use *reduction triples* [24,31].

Definition 48 (Reduction triple). *A* reduction triple $(\succsim, \succcurlyeq, \succ)$ *consists of two quasi-orderings* $\succsim$ *and* $\succcurlyeq$ *and a well-founded strict ordering* $\succ$ *on meta-terms such that* $\succsim$ *is monotonic, all of* $\succsim, \succcurlyeq, \succ$ *are meta-stable (that is,* $\ell \succsim r$ *implies* $\ell\gamma \succsim r\gamma$ *if* ℓ *is a closed pattern and* γ *a substitution on domain* $FMV(\ell) \cup FMV(r)$*, and the same for* $\succcurlyeq$ *and* $\succ$*),* $\Rightarrow_\beta \subseteq \succsim$*, and both* $\succsim \circ \succ \subseteq \succ$ *and* $\succcurlyeq \circ \succ \subseteq \succ$*.*

In the first-order DP framework, the reduction pair processor [20] seeks to orient all rules with $\succsim$ and all DPs with either $\succsim$ or $\succ$; if this succeeds, those pairs oriented with $\succ$ may be removed. Using reduction *triples* rather than pairs, we obtain the following extension to the higher-order setting:

Theorem 49 (Basic reduction triple processor). *Let* $M = (\mathcal{P}_1 \uplus \mathcal{P}_2, \mathcal{R}, m, f)$ *be a DP problem. If* $(\succsim, \succcurlyeq, \succ)$ *is a reduction triple such that*

1. *for all* $\ell \Rightarrow r \in \mathcal{R}$*, we have* $\ell \succsim r$*;*
2. *for all* $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1$*, we have* $\ell \succ p$*;*
3. *for all* $\ell \Rrightarrow p\ (A) \in \mathcal{P}_2$*, we have* $\ell \succcurlyeq p$*;*

then the processor that maps M *to* $\{(\mathcal{P}_2, \mathcal{R}, m, f)\}$ *is both sound and complete.*

Proof (sketch). For an infinite $(\mathcal{P}_1 \uplus \mathcal{P}_2, \mathcal{R})$-chain $[(\rho_0, s_0, t_0), (\rho_1, s_1, t_1), \ldots]$ the requirements provide that, for all i: (a) $s_i \succ t_i$ if $\rho_i \in \mathcal{P}_1$; (b) $s_i \succcurlyeq t_i$ if $\rho_i \in \mathcal{P}_2$; and (c) $t_i \succsim s_{i+1}$. Since $\succ$ is well-founded, only finitely many DPs can be in $\mathcal{P}_1$, so a tail of the chain is actually an infinite $(\mathcal{P}_2, \mathcal{R}, m, f)$-chain. □

Example 50. Let $(\mathcal{F}, \mathcal{R})$ be the (non-η-expanded) rules from Ex. 17, and $SDP(\mathcal{R})$ the DPs from Ex. 28. From Thm. 49, we get the following ordering requirements:

$$\begin{aligned} \mathtt{deriv}\ (\lambda x.\mathtt{sin}\ F\langle x\rangle) &\succsim \lambda y.\mathtt{times}\ (\mathtt{deriv}\ (\lambda x.F\langle x\rangle)\ y)\ (\mathtt{cos}\ F\langle y\rangle) \\ \mathtt{deriv}^\sharp\ (\lambda x.\mathtt{sin}\ F\langle x\rangle) &\succ \mathtt{deriv}^\sharp\ (\lambda x.F\langle x\rangle) \end{aligned}$$

We can handle both requirements by using a polynomial interpretation $\mathcal{J}$ to $\mathbb{N}$ [43,15], by choosing $\mathcal{J}_{\mathtt{sin}}(n) = n{+}1$, $\mathcal{J}_{\mathtt{cos}}(n) = 0$, $\mathcal{J}_{\mathtt{times}}(n_1, n_2) = n_1$, $\mathcal{J}_{\mathtt{deriv}}(f) = \mathcal{J}_{\mathtt{deriv}^\sharp}(f) = \lambda n.f(n)$. Then the requirements are evaluated to: $\lambda n.f(n) + 1 \geq \lambda n.f(n)$ and $\lambda n.f(n) + 1 > \lambda n.f(n)$, which holds on $\mathbb{N}$.

Thm. 49 is not ideal since, by definition, the left- and right-hand side of a DP may have different types. Such DPs are hard to handle with traditional techniques such as HORPO [26] or polynomial interpretations [43,15], as these methods compare only (meta-)terms of the same type (modulo renaming of sorts).

Example 51. Consider the toy AFSM with $\mathcal{R} = \{\mathtt{f}\ (\mathtt{s}\ X)\ Y \Rightarrow \mathtt{g}\ X\ Y,\ \mathtt{g}\ X \Rightarrow \lambda z.\mathtt{f}\ X\ z\}$ and $SDP(\mathcal{R}) = \{\mathtt{f}^\sharp\ (\mathtt{s}\ X)\ Y \Rrightarrow \mathtt{g}^\sharp\ X,\ \mathtt{g}^\sharp\ X \Rrightarrow \mathtt{f}^\sharp\ X\ Z\}$. If $\mathtt{f}$ and $\mathtt{g}$ both have a type $\mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$, then in the first DP, the left-hand side has type $\mathtt{nat}$ while the right-hand side has type $\mathtt{nat} \to \mathtt{nat}$. In the second DP, the left-hand side has type $\mathtt{nat} \to \mathtt{nat}$ and the right-hand side has type $\mathtt{nat}$.

To be able to handle examples like the one above, we adapt [31, Thm. 5.21] by altering the ordering requirements to have base type.

Theorem 52 (Reduction triple processor). *Let* Bot *be a set* $\{\bot_\sigma : \sigma \mid \sigma \text{ a type}\} \subseteq \mathcal{F}^\sharp$ *of unused constructors,* $M = (\mathcal{P}_1 \uplus \mathcal{P}_2, \mathcal{R}, m, f)$ *a DP problem and* $(\succsim, \succcurlyeq, \succ)$ *a reduction triple such that: (a) for all* $\ell \Rightarrow r \in \mathcal{R}$*, we have* $\ell \succsim r$*; and (b) for all* $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1 \uplus \mathcal{P}_2$ *with* $\ell : \sigma_1 \to \ldots \to \sigma_m \to \iota$ *and* $p : \tau_1 \to \ldots \to \tau_n \to \kappa$ *we have, for fresh meta-variables* $Z_1 : \sigma_1, \ldots, Z_m : \sigma_m$*:*

- $\ell\ Z_1 \cdots Z_m \succ p\ \bot_{\tau_1} \cdots \bot_{\tau_n}$ *if* $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1$
- $\ell\ Z_1 \cdots Z_m \succcurlyeq p\ \bot_{\tau_1} \cdots \bot_{\tau_n}$ *if* $\ell \Rrightarrow p\ (A) \in \mathcal{P}_2$

Then the processor that maps M *to* $\{(\mathcal{P}_2, \mathcal{R}, m, f)\}$ *is both sound and complete.*

Proof (sketch). If $(\succsim, \succcurlyeq, \succ)$ is such a triple, then for $R \in \{\succcurlyeq, \succ\}$ define R' as follows: for $s : \sigma_1 \to \ldots \to \sigma_m \to \iota$ and $t : \tau_1 \to \ldots \to \tau_n \to \kappa$, let $s\ R'\ t$ if for all $u_1 : \sigma_1, \ldots, u_m : \sigma_m$ there exist $w_1 : \tau_1, \ldots, w_n : \tau_n$ such that $s\ u_1 \cdots u_m\ R\ t\ w_1 \cdots w_n$. Now apply Thm. 49 with the triple $(\succsim, \succcurlyeq', \succ')$. □

Here, the elements of Bot take the role of minimal terms for the ordering. We use them to flatten the type of the right-hand sides of ordering requirements, which makes it easier to use traditional methods to generate a reduction triple.

While $\succ$ and $\succcurlyeq$ may still have to orient meta-terms of distinct types, these are always *base* types, which we could collapse to a single sort. The only relation required to be monotonic, $\succsim$, regards pairs of meta-terms of the *same* type. This makes it feasible to apply orderings like HORPO or polynomial interpretations.

Both the basic and non-basic reduction triple processor are difficult to use for *non-conservative* DPs, which generate ordering requirements whose right-hand side contains a meta-variable not occurring on the left. This is typically difficult for traditional techniques, although possible to overcome, by choosing triples that do not regard such meta-variables (e.g., via an argument filtering [35,46]):

Example 53. We apply Thm. 52 on the DP problem $(\mathit{SDP}(\mathcal{R}), \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative})$ of Ex. 51. This gives for instance the following ordering requirements:

$$\begin{array}{rlrl} \mathtt{f}\ (\mathtt{s}\ X)\ Y \succsim \mathtt{g}\ X\ Y & & \mathtt{f}^\sharp\ (\mathtt{s}\ X)\ Y \succ \mathtt{g}^\sharp\ X\ \bot_{\mathtt{nat}} \\ \mathtt{g}\ X \succsim \lambda z.\mathtt{f}\ X\ z & & \mathtt{g}^\sharp\ X\ Y \succcurlyeq \mathtt{f}^\sharp\ X\ Z \end{array}$$

The right-hand side of the last DP uses a meta-variable Z that does not occur on the left. As neither $\succ$ nor $\succcurlyeq$ are required to be monotonic (only $\succsim$ is), function symbols do not have to regard all their arguments. Thus, we can use a polynomial interpretation $\mathcal{J}$ to $\mathbb{N}$ with $\mathcal{J}_{\bot_{\mathtt{nat}}} = 0$, $\mathcal{J}_{\mathtt{s}}(n) = n + 1$ and $\mathcal{J}_{\mathtt{h}}(n_1, n_2) = n_1$ for $\mathtt{h} \in \{\mathtt{f}, \mathtt{f}^\sharp, \mathtt{g}, \mathtt{g}^\sharp\}$. The ordering requirements then translate to $X + 1 \geq X$ and $\lambda y.X \geq \lambda z.X$ for the rules, and $X + 1 > X$ and $X \geq X$ for the DPs. All these inequalities on $\mathbb{N}$ are clearly satisfied, so we can remove the first DP. The remaining problem is quickly dispersed with the dependency graph processor.

5.3 Rule removal without search for orderings

While processors often simplify only $\mathcal{P}$, they can also simplify $\mathcal{R}$. One of the most powerful techniques in first-order DP approaches that can do this are *usable rules*. The idea is that for a given set $\mathcal{P}$ of DPs, we only need to consider a *subset* $\mathit{UR}(\mathcal{P}, \mathcal{R})$ of $\mathcal{R}$. Combined with the dependency graph processor, this makes it possible to split a large term rewriting system into a number of small problems.

In the higher-order setting, simple versions of usable rules have also been defined [46,31]. We can easily extend these definitions to AFSMs:

Theorem 54. *Given a DP problem $M = (\mathcal{P}, \mathcal{R}, m, f)$ with $m \succeq$ `minimal` and $\mathcal{R}$ finite, let $UR(\mathcal{P}, \mathcal{R})$ be the smallest subset of $\mathcal{R}$ such that:*

- *if a symbol* `f` *occurs in the right-hand side of an element of $\mathcal{P}$ or $UR(\mathcal{P}, \mathcal{R})$, and there is a rule $\mathtt{f}\ \ell_1 \cdots \ell_k \Rightarrow r$, then this rule is also in $UR(\mathcal{P}, \mathcal{R})$;*
- *if there exists $\ell \Rightarrow r \in \mathcal{R}$ or $\ell \Rrightarrow r\ (A) \in \mathcal{P}$ such that $r \trianglerighteq F\langle s_1, \dots, s_k\rangle\ t_1 \cdots t_n$ with $s_1, \dots, s_k$ not all distinct variables or with $n > 0$, then $UR(\mathcal{P}, \mathcal{R}) = \mathcal{R}$.*

Then the processor that maps M to $\{(\mathcal{P}, UR(\mathcal{P}, \mathcal{R}), \mathtt{arbitrary}, \mathtt{all})\}$ is sound.

For the proof we refer to the very similar proofs in [46,31].

Example 55. For the set $SDP(\mathcal{R})$ of the ordinal recursion example (Ex. 8 and 29), all rules are usable due to the occurrence of $H\ M$ in the second DP. For the set $SDP(\mathcal{R})$ of the map example (Ex. 6 and 31), there are no usable rules, since the one DP contains no defined function symbols or applied meta-variables.

This higher-order processor is much less powerful than its first-order version: if any DP or usable rule has a sub-meta-term of the form $F\ s$ or $F\langle s_1, \dots, s_k\rangle$ with $s_1, \dots, s_k$ not all distinct variables, then *all* rules are usable. Since applying a higher-order meta-variable to some argument is extremely common in higher-order rewriting, the technique is usually not applicable. Also, this processor imposes a heavy price on the flags: minimality (at least) is required, but is lost; the formative flag is also lost. Thus, usable rules are often combined with reduction triples to temporarily disregard rules, rather than as a way to permanently remove rules.

To address these weaknesses, we consider a processor that uses similar ideas to usable rules, but operates from the *left-hand* sides of rules and DPs rather than the right. This adapts the technique from [31] that relies on the new *formative* flag. As in the first-order case [16], we use a semantic characterisation of formative rules. In practice, we then work with over-approximations of this characterisation, analogous to the use of dependency graph approximations in Thm. 45.

Definition 56. *A function FR that maps a pattern ℓ and a set of rules $\mathcal{R}$ to a set $FR(\ell, \mathcal{R}) \subseteq \mathcal{R}$ is a* formative rules approximation *if for all s and γ: if $s \Rightarrow^*_{\mathcal{R}} \ell\gamma$ by an ℓ-formative reduction, then this reduction can be done using only rules in $FR(\ell, \mathcal{R})$.*

We let $FR(\mathcal{P}, \mathcal{R}) = \bigcup\{FR(\ell_i, \mathcal{R}) \mid \mathtt{f}\ \ell_1 \cdots \ell_n \Rrightarrow p\ (A) \in \mathcal{P} \wedge 1 \leq i \leq n\}$.

Thus, a formative rules approximation is a subset of $\mathcal{R}$ that is *sufficient* for a formative reduction: if $s \Rightarrow^*_{\mathcal{R}} \ell\gamma$, then $s \Rightarrow^*_{FR(\ell,\mathcal{R})} \ell\gamma$. It is allowed for there to exist other formative reductions that do use additional rules.

Example 57. We define a simple formative rules approximation: (1) $FR(Z, \mathcal{R}) = \emptyset$ if Z is a meta-variable; (2) $FR(\mathtt{f}\ \ell_1 \cdots \ell_m, \mathcal{R}) = FR(\ell_1, \mathcal{R}) \cup \cdots \cup FR(\ell_m, \mathcal{R})$ if $\mathtt{f} : \sigma_1 \rightarrow \ldots \rightarrow \sigma_m \rightarrow \iota$ and no rules have type ι; (3) $FR(s, \mathcal{R}) = \mathcal{R}$ otherwise. This is a formative rules approximation: if $s \Rightarrow^*_{\mathcal{R}} Z\gamma$ by a Z-formative reduction, then $s = Z\gamma$, and if $s \Rightarrow^*_{\mathcal{R}} \mathtt{f}\ \ell_1 \cdots \ell_m$ and no rules have the same output type as s, then $s = \mathtt{f}\ s_1 \cdots s_m$ and each $s_i \Rightarrow^*_{\mathcal{R}} \ell_i\gamma$ (by an ℓ_i-formative reduction).

The following result follows directly from the definition of formative rules.

Theorem 58 (Formative rules processor). *For a formative rules approximation FR, the processor $Proc_{FR}$ that maps a DP problem $(\mathcal{P}, \mathcal{R}, m, \texttt{formative})$ to $\{(\mathcal{P}, FR(\mathcal{P}, \mathcal{R}), m, \texttt{formative})\}$ is both sound and complete.*

Proof (sketch). A processor that only removes rules (or DPs) is always complete. For soundness, if the chain is formative then each step $t_i \Rrightarrow^*_{\mathcal{R}} s_{i+1}$ can be replaced by $t_i \Rrightarrow^*_{FR(\mathcal{P},\mathcal{R})} s_{i+1}$. Thus, the chain can be seen as a $(\mathcal{P}, FR(\mathcal{P}, \mathcal{R}))$-chain. $\square$

Example 59. For our ordinal recursion example (Ex. 8 and 29), *none* of the rules are included when we use the approximation of Ex. 57 since all rules have output type `ord`. Thus, $Proc_{FR}$ maps $(SDP(\mathcal{R}), \mathcal{R}, \texttt{computable}_{\mathcal{R}}, \texttt{formative})$ to $(SDP(\mathcal{R}), \emptyset, \texttt{computable}_{\mathcal{R}}, \texttt{formative})$. *Note:* this example can also be completed without formative rules (see Ex. 64). Here we illustrate that, even with a simple formative rules approximation, we can often delete all rules of a given type.

Formative rules are introduced in [31], and the definitions can be adapted to a more powerful formative rules approximation than the one sketched in Ex. 59. Several examples and deeper intuition for the first-order setting are given in [16].

5.4 Subterm criterion processors

Reduction triple processors are powerful, but they exert a computational price: we must orient all rules in $\mathcal{R}$. The subterm criterion processor allows us to remove DPs without considering $\mathcal{R}$ at all. It is based on a *projection function* [24], whose higher-order counterpart [34,46,31] is the following:

Definition 60. *For $\mathcal{P}$ a set of DPs, let $\texttt{heads}(\mathcal{P})$ be the set of all symbols $\texttt{f}$ that occur as the head of a left- or right-hand side of a DP in $\mathcal{P}$. A* projection function *for $\mathcal{P}$ is a function $\nu : \texttt{heads}(\mathcal{P}) \to \mathbb{N}$ such that for all DPs $\ell \Rrightarrow p\ (A) \in \mathcal{P}$, the function $\overline{\nu}$ with $\overline{\nu}(\texttt{f}\ s_1 \cdots s_n) = s_{\nu(\texttt{f})}$ is well-defined both for ℓ and for p.*

Theorem 61 (Subterm criterion processor). *The processor $Proc_{\texttt{subcrit}}$ that maps a DP problem $(\mathcal{P}_1 \uplus \mathcal{P}_2, \mathcal{R}, m, f)$ with $m \succeq \texttt{minimal}$ to $\{(\mathcal{P}_2, \mathcal{R}, m, f)\}$ if a projection function ν exists such that $\overline{\nu}(\ell) \rhd \overline{\nu}(p)$ for all $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1$ and $\overline{\nu}(\ell) = \overline{\nu}(p)$ for all $\ell \Rrightarrow p\ (A) \in \mathcal{P}_2$, is sound and complete.*

Proof (sketch). If the conditions are satisfied, every infinite $(\mathcal{P}, \mathcal{R})$-chain induces an infinite $\unrhd \cdot \Rrightarrow^*_{\mathcal{R}}$ sequence that starts in a strict subterm of t_1, contradicting minimality unless all but finitely many steps are equality. Since every occurrence of a pair in $\mathcal{P}_1$ results in a strict $\rhd$ step, a tail of the chain lies in $\mathcal{P}_2$. $\square$

Example 62. Using $\nu(\texttt{map}^\sharp) = 2$, $Proc_{\texttt{subcrit}}$ maps the DP problem $(\{(1)\}, \mathcal{R}, \texttt{computable}_{\mathcal{R}}, \texttt{formative})$ from Ex. 47 to $\{(\emptyset, \mathcal{R}, \texttt{computable}_{\mathcal{R}}, \texttt{formative})\}$.

The subterm criterion can be strengthened, following [34,46], to also handle DPs like the one in Ex. 28. Here, we focus on a new idea. For *computable* chains, we can build on the idea of the subterm criterion to get something more.

Theorem 63 (Computable subterm criterion processor). *The processor $Proc_{\texttt{statcrit}}$ that maps a DP problem $(P_1 \uplus \mathcal{P}_2, \mathcal{R}, \texttt{computable}_{\mathcal{U}}, f)$ to $\{(\mathcal{P}_2, \mathcal{R}, \texttt{computable}_{\mathcal{U}}, f)\}$ if a projection function ν exists such that $\overline{\nu}(\ell) \sqsupset \overline{\nu}(p)$ for all $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1$ and $\overline{\nu}(\ell) = \overline{\nu}(p)$ for all $\ell \Rrightarrow p\ (A) \in \mathcal{P}_2$, is sound and complete. Here, $\sqsupset$ is the relation on base-type terms with $s \sqsupset t$ if $s \neq t$ and (a) $s \trianglerighteq_{\texttt{acc}} t$ or (b) a meta-variable Z exists with $s \trianglerighteq_{\texttt{acc}} Z\langle x_1, \dots, x_k\rangle$ and $t = Z\langle t_1, \dots, t_k\rangle\ s_1 \cdots s_n$.*

Proof (sketch). By the conditions, every infinite $(\mathcal{P}, \mathcal{R})$-chain induces an infinite $(\Rrightarrow_{C_{\mathcal{U}}} \cup \Rightarrow_\beta)^* \cdot \Rightarrow_{\mathcal{R}}^*$ sequence (where $C_{\mathcal{U}}$ is defined following Thm. 13). This contradicts computability unless there are only finitely many inequality steps. As pairs in $\mathcal{P}_1$ give rise to a strict decrease, they may occur only finitely often. □

Example 64. Following Ex. 8 and 29, consider the projection function ν with $\nu(\texttt{rec}^\sharp) = 1$. As $\texttt{s}\ X \trianglerighteq_{\texttt{acc}} X$ and $\texttt{lim}\ H \trianglerighteq_{\texttt{acc}} H$, both $\texttt{s}\ X \sqsupset X$ and $\texttt{lim}\ H \sqsupset H\ M$ hold. Thus $Proc_{\texttt{statc}}(\mathcal{P}, \mathcal{R}, \texttt{computable}_{\mathcal{R}}, \texttt{formative}) = \{(\emptyset, \mathcal{R}, \texttt{computable}_{\mathcal{R}}, \texttt{formative})\}$. By the dependency graph processor, the AFSM is terminating.

The computable subterm criterion processor fundamentally relies on the new $\texttt{computable}_{\mathcal{U}}$ flag, so it has no counterpart in the literature so far.

5.5 Non-termination

While (most of) the processors presented so far are complete, none of them can actually return NO. We have not yet implemented such a processor; however, we can already provide a general specification of a *non-termination processor.*

Theorem 65 (Non-termination processor). *Let $M = (\mathcal{P}, \mathcal{R}, m, f)$ be a DP problem. The processor that maps M to NO if it determines that a sufficient criterion for non-termination of $\Rightarrow_{\mathcal{R}}$ or for existence of an infinite conservative $(\mathcal{P}, \mathcal{R})$-chain according to the flags m and f holds is sound and complete.*

Proof. Obvious. □

This is a very general processor, which does not tell us *how* to determine such a sufficient criterion. However, it allows us to conclude non-termination as part of the framework by identifying a suitable infinite chain.

Example 66. If we can find a finite $(\mathcal{P}, \mathcal{R})$-chain $[(\rho_0, s_0, t_0), \dots, (\rho_n, s_n, t_n)]$ with $t_n = s_0\gamma$ for some substitution γ which uses only conservative DPs, is formative if $f = \texttt{formative}$ and is $\mathcal{U}$-computable if $m = \texttt{computable}_{\mathcal{U}}$, such a chain is clearly a sufficient criterion: there is an infinite chain $[(\rho_0, s_0, t_0), \dots, (\rho_0, s_0\gamma, t_0\gamma), \dots, (\rho_0, s_0\gamma\gamma, t_0\gamma\gamma), \dots]$. If $m = \texttt{minimal}$ and we find such a chain that is however not minimal, then note that $\Rightarrow_{\mathcal{R}}$ is non-terminating, which also suffices.

For example, for a DP problem $(\mathcal{P}, \mathcal{R}, \texttt{minimal}, \texttt{all})$ with $\mathcal{P} = \{\texttt{f}^\sharp\ F\ X \Rrightarrow \texttt{g}^\sharp\ (F\ X),\ \texttt{g}^\sharp\ X \Rrightarrow \texttt{f}^\sharp\ \texttt{h}\ X\}$, there is a finite dependency chain: $[(\texttt{f}^\sharp\ F\ X \Rrightarrow \texttt{g}^\sharp\ (F\ X),\ \texttt{f}^\sharp\ \texttt{h}\ x,\ \texttt{g}^\sharp\ (\texttt{h}\ x)),\ (\texttt{g}^\sharp\ X \Rrightarrow \texttt{f}^\sharp\ \texttt{h}\ X,\ \texttt{g}^\sharp\ (\texttt{h}\ x),\ \texttt{f}^\sharp\ \texttt{h}\ (\texttt{h}\ x))]$. As $\texttt{f}^\sharp\ \texttt{h}\ (\texttt{h}\ x)$ is an instance of $\texttt{f}^\sharp\ \texttt{h}\ x$, the processor maps this DP problem to NO.

To instantiate Thm. 65, we can borrow non-termination criteria from first-order rewriting [21,42,13], with minor adaptions to the typed setting. Of course, it is worthwhile to also investigate dedicated higher-order non-termination criteria.

6 Conclusions and Future Work

We have built on the static dependency pair approach [6,34,46,33] and formulated it in the language of the DP *framework* from first-order rewriting [20,22]. Our formulation is based on AFSMs, a dedicated formalism designed to make termination proofs transferrable to various higher-order rewriting formalisms.

This framework has two important additions over existing higher-order DP approaches in the literature. First, we consider not only arbitrary and minimally non-terminating dependency chains, but also minimally *non-computable* chains; this is tracked by the $\mathtt{computable}_{\mathcal{U}}$ flag. Using the flag, a dedicated processor allows us to efficiently handle rules like Ex. 8. This flag has no counterpart in the first-order setting. Second, we have generalised the idea of formative rules in [31] to a notion of formative *chains*, tracked by a `formative` flag. This makes it possible to define a corresponding processor that permanently removes rules.

Implementation and experiments. To provide a strong formal groundwork, we have presented several processors in a general way, using semantic definitions of, e.g., the dependency graph approximation and formative rules rather than syntactic definitions using functions like *TCap* [21]. Even so, most parts of the DP framework for AFSMs have been implemented in the open-source termination prover WANDA [28], alongside a dynamic DP framework [31] and a mechanism to delegate some ordering constraints to a first-order tool [14]. For reduction triples, polynomial interpretations [15] and a version of HORPO [29, Ch. 5] are used. To solve the constraints arising in the search for these orderings, and also to determine sort orderings (for the accessibility relation) and projection functions (for the subterm criteria), WANDA employs an external SAT-solver. WANDA has won the higher-order category of the International Termination Competition [50] four times. In the International Confluence Competition [10], the tools ACPH [40] and CSI^ho [38] use WANDA as their "oracle" for termination proofs on HRSs.

We have tested WANDA on the *Termination Problems Data Base* [49], using AProVE [19] and MiniSat [12] as back-ends. When no additional features are enabled, WANDA proves termination of 124 (out of 198) benchmarks with static DPs, versus 92 with only a search for reduction orderings; a 34% increase. When all features except static DPs are enabled, WANDA succeeds on 153 benchmarks, versus 166 with also static DPs; an 8% increase, or alternatively, a 29% decrease in failure rate. The full evaluation is available in Appendix D.

Future work. While the static and the dynamic DP approaches each have their own strengths, there has thus far been little progress on a *unified* approach, which could take advantage of the syntactic benefits of both styles. We plan to combine the present work with the ideas of [31] into such a unified DP framework.

In addition, we plan to extend the higher-order DP framework to rewriting with *strategies*, such as implicit β-normalisation or strategies inspired by functional programming languages like OCaml and Haskell. Other natural directions are dedicated automation to detect non-termination, and reducing the number of term constraints solved by the reduction triple processor via a tighter integration with usable and formative rules with respect to argument filterings.

References

1. P. Aczel. A general Church-Rosser theorem. Unpublished Manuscript, University of Manchester, 1978.
2. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.
3. F. Baader and F. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
4. L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.
5. F. Blanqui. Termination and confluence of higher-order rewrite systems. In *Proc. RTA '00*, 2000.
6. F. Blanqui. Higher-order dependency pairs. In *Proc. WST '06*, 2006.
7. F. Blanqui. Termination of rewrite relations on λ-terms based on Girard's notion of reducibility. *Theoretical Computer Science*, 611:50–86, 2016.
8. F. Blanqui, J. Jouannaud, and M. Okada. Inductive-data-type systems. *Theoretical Computer Science*, 272(1-2):41–68, 2002.
9. F. Blanqui, J. Jouannaud, and A. Rubio. The computability path ordering. *Logical Methods in Computer Science*, 11(4), 2015.
10. Community. The international Confluence Competition (CoCo). `http://project-coco.uibk.ac.at/`, 2018.
11. N. Dershowitz and S. Kaplan. Rewrite, rewrite, rewrite, rewrite, rewrite. In *Proc. POPL '89*, 1989.
12. N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. SAT '03*, pages 502–518, 2004. See also `http://minisat.se/`.
13. F. Emmes, T. Enger, and J. Giesl. Proving non-looping non-termination automatically. In *Proc. IJCAR '12*, 2012.
14. C. Fuhs and C. Kop. Harnessing first order termination provers using higher order dependency pairs. In *Proc. FroCoS '11*, 2011.
15. C. Fuhs and C. Kop. Polynomial interpretations for higher-order rewriting. In *Proc. RTA '12*, 2012.
16. C. Fuhs and C. Kop. First-order formative rules. In *Proc. RTA-TLCA '14*, 2014.
17. C. Fuhs and C. Kop. A static higher-order dependency pair framework. In *Proc. ESOP'19*, 2019. To appear.
18. C. Fuhs, C. Kop, and N. Nishida. Verifying procedural programs via constrained rewriting induction. *ACM Transactions on Computational Logic*, 18(2):14:1–14:50, 2017.
19. J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 2017.
20. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR '04*, 2005.
21. J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. FroCoS '05*, 2005.
22. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.
23. F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *Proc. FLOPS '10*, 2010.

24. N. Hirokawa and A. Middeldorp. Tyrolean termination tool: Techniques and features. *Information and Computation*, 205(4):474–511, 2007.
25. J. C. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *Proc. VLSI '99*, 1999.
26. J. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proc. LICS '99*, 1999.
27. J. Klop, V. v. Oostrom, and F. v. Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1-2):279–308, 1993.
28. C. Kop. WANDA – a higher-order termination tool. `http://wandahot.sourceforge.net/`.
29. C. Kop. *Higher Order Termination*. PhD thesis, VU Amsterdam, 2012.
30. C. Kop and F. v. Raamsdonk. Higher order dependency pairs for algebraic functional systems. In *Proc. RTA '11*, 2011.
31. C. Kop and F. v. Raamsdonk. Dynamic dependency pairs for algebraic functional systems. *Logical Methods in Computer Science*, 8(2):10:1–10:51, 2012.
32. K. Kusakari. Static dependency pair method in rewriting systems for functional programs with product, algebraic data, and ML-polymorphic types. *IEICE Transactions*, 96-D(3):472–480, 2013.
33. K. Kusakari. Static dependency pair method in functional programs. *IEICE Transactions on Information and Systems*, E101.D(6):1491–1502, 2018.
34. 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.
35. K. Kusakari, M. Nakamura, and Y. Toyama. Argument filtering transformation. In *Proc. PPDP '99*, 1999.
36. C. A. Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1(1):5–36, 1992.
37. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
38. J. Nagele. CoCo 2018 participant: CSIˆho 0.2. `http://project-coco.uibk.ac.at/2018/papers/csiho.pdf`, 2018.
39. T. Nipkow. Higher-order critical pairs. In *Proc. LICS '91*, 1991.
40. K. Onozawa, K. Kikuchi, T. Aoto, and Y. Toyama. ACPH: System description for CoCo 2017. `http://project-coco.uibk.ac.at/2017/papers/acph.pdf`, 2017.
41. C. Otto, M. Brockschmidt, C. v. Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *Proc. RTA '10*, 2010.
42. É. Payet. Loop detection in term rewriting using the eliminating unfoldings. *Theoretical Computer Science*, 403(2-3):307–327, 2008.
43. J. v. d. Pol. *Termination of Higher-order Rewrite Systems*. PhD thesis, University of Utrecht, 1996.
44. 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.
45. 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.
46. 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.
47. W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32(2):187–199, 1967.

48. Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.
49. Wiki. Termination Problems DataBase (tpdb). `http://termination-portal.org/wiki/TPDB`.
50. Wiki. The International Termination Competition (TermComp). `http://termination-portal.org/wiki/Termination_Competition`, 2018.

– APPENDIX –

This appendix contains detailed proofs for all results in the paper. Proofs in higher-order rewriting are typically intricate and subject to errors in the small details, so we have strived to be very precise. However, aside from Appendix A (which is simply an adaptation of an existing technique to the present setting), the main idea of all proofs is captured by the proof sketches in the paper.

In addition, Appendix D presents an experimental evaluation that considers the power of the techniques in this paper on the termination problem database [49].

A Computability: the set C

In this appendix, we prove Thm. 13: the existence of an RC-set C that provides an accessibility relation $\trianglerighteq_{\mathtt{acc}}$ that preserves computability, and a base-type accessibility step $\Rrightarrow_C$ that preserves both computability and termination.

As we have said before, $\mathcal{V}$ and $\mathcal{F}$ contain infinitely many symbols of all types. We will use this to select variables or constructor symbol of any given type without further explanation.

These proofs *do not require* that computability is considered with respect to a rewrite relation: other relations (such as recursive path orderings) may be used as well. To make this explicit, we will use an alternative relation symbol, $\sqsupset$.

The proofs here consider a computability notion over the set $\mathcal{T}(\mathcal{F},\mathcal{V})$ of terms without restrictions. However, they could easily be extended to subsets of a different set of terms T, provided T is closed under $\Rightarrow_{\mathcal{R}}$. This could for instance be used to obtain a computability result for terms that satisfy certain arity restrictions. To make this generality clear, each quantification over terms is explicitly marked with $\mathcal{T}(\mathcal{F},\mathcal{V})$.

Note: a more extensive discussion of computability can be found in [7]. Our notion of accessibility largely corresponds to membership of the computability closure defined there (although not completely).

A.1 Definitions and main computability result

Definition 67. *In Appendix A, $\sqsupset$ is assumed to be a given relation on terms of the same type, with respect to which we consider computability. We require that:*

- *$\sqsupset$ is monotonic (that is, $s \sqsupset t$ implies that $s\ u \sqsupset t\ u$ and $u\ s \sqsupset u\ t$ and $\lambda x.s \sqsupset \lambda x.t$);*
- *for all variables x: $x\ s_1 \cdots s_n \sqsupset t$ implies that t has the form $x\ s_1 \cdots s_i' \cdots s_n$ with $s_i \sqsupset s_i'$;*
- *if $s \Rightarrow^*_{\mathtt{head}\beta} u$ and $s \sqsupset t$, then there exists v such that $u \sqsupset^* v$ and $t \Rightarrow^*_{\mathtt{head}\beta} v$; here, $\Rightarrow_{\mathtt{head}\beta}$ is the relation generated by the step $(\lambda x.u)\ v\ w_1 \cdots w_n \Rightarrow_{\mathtt{head}\beta} u[x := v]\ w_1 \cdots w_n$;*
- *if t is the $\mathtt{head}\beta$-normal form of s, then $s \sqsupset^* t$.*

We call a term neutral *if it has the form $x\ s_1 \cdots s_n$ or $(\lambda x.u)\ s_0 \cdots s_n$.*

The generality obtained by imposing only the minimal requirements on $\sqsupset$ is not needed in the current paper (where we only consider computability with respect to a rewrite relation), but could be used to extend the method to other domains. First note:

Lemma 68. *A rewrite relation $\Rightarrow_{\mathcal{R}}$ satisfies the requirements of $\sqsupset$ stated in Def. 67.*

Proof. Clearly $\Rightarrow_{\mathcal{R}}$ is monotonic, applications with a variable at the head cannot be reduced at the head, and moreover $\Rightarrow_{\mathcal{R}}$ includes $\Rightarrow_{\mathtt{head}\beta}$.

The third property we prove by induction on s with $\Rightarrow_\beta$, using $\Rightarrow_{\mathcal{R}}^*$ instead of $\Rightarrow_{\mathcal{R}}$ for a stronger induction hypothesis. If $s = u$, then we are done choosing $v := t$. Otherwise we can write $s = (\lambda x.q)\ w_0\ w_1 \cdots w_n$ and $s \Rightarrow_{\mathtt{head}\beta} s' := q[x := w_0]\ w_1 \cdots w_n$, and $s' \Rightarrow_{\mathtt{head}\beta}^* u$. If the reduction $s \Rightarrow_{\mathcal{R}}^* t$ does not take any head steps, then

$$t = (\lambda x.q')\ w_0'\ w_1' \cdots w_n' \Rightarrow_{\mathtt{head}\beta}^* q'[x := w_0']\ w_1' \cdots w_n' =: v$$

and indeed $u \Rightarrow_{\mathcal{R}}^* v$ by monotonicity. Otherwise, by the same argument we can safely assume that the head step is done first, so $s' \Rightarrow_{\mathcal{R}}^* t$; we complete by the induction hypothesis. □

Recall Def. 10 from the text.

Definition 10 (with $\sqsupset$ rather than $\Rightarrow_{\mathcal{R}}$). *A* set of reducibility candidates*, or* RC-set*, for a relation $\sqsupset$ as in Def. 67 is a set I of base-type terms $s \in \mathcal{T}(\mathcal{F},\mathcal{V})$ such that:*

- *every term in I is terminating under $\sqsupset$*
- *I is closed under $\sqsupset$ (so if $s \in I$ and $s \sqsupset t$ then $t \in I$)*
- *if $s \in \mathcal{T}(\mathcal{F},\mathcal{V})$ is neutral, and for all t with $s \sqsupset t$ we have $t \in I$, then $s \in I$*

We define I-computability for an RC-set I by induction on types; for $s \in \mathcal{T}(\mathcal{F},\mathcal{V})$ we say s is I-computable if:

- *$s : \iota$ for some $\iota \in \mathcal{S}$ and $s \in I$ ($\iota \in \mathcal{S}$)*
- *$s : \sigma \to \tau$ and for all terms $t : \sigma \in \mathcal{T}(\mathcal{F},\mathcal{V})$ that are I-computable, $s\ t$ is I-computable*

For ι a sort and I an RC-set, we will write $I(\iota) = \{s \in I \mid s : \iota\}$.

Let us illustrate Def. 10 with two examples:

Lemma 69. *The set SN of all terminating base-type terms in $\mathcal{T}(\mathcal{F},\mathcal{V})$ is an RC-set. The set MIN of all terminating base-type terms in $\mathcal{T}(\mathcal{F},\mathcal{V})$ whose* $\mathtt{head}\beta$*-normal form can be written $x\ s_1 \cdots s_m$ with $x \in \mathcal{V}$ is also an RC-set.*

Proof. It is easy to verify that the requirements hold for SN. For MIN, clearly termination holds. If $s \in \mathsf{MIN}$, then $s \Rightarrow_{\mathtt{head}\beta}^* x\ s_1 \cdots s_m =: s'$, so for any t with $s \sqsupset^* t$ the assumptions on $\sqsupset$ provide that $t \Rightarrow_{\mathtt{head}\beta}^* v$ for some $\sqsupset^*$-reduct of

s', which can only have the form $x\ t_1 \cdots t_m$. Finally, we prove that a neutral term $s \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ is in MIN if all its $\sqsupset^+$-reducts are, by induction on s with $\Rightarrow_\beta$ (this suffices because we have already seen that MIN is closed under $\sqsupset$). If $s = x\ s_1 \cdots s_m$ then it is included in MIN if it is terminating, which is the case if all its reducts are terminating, which is certainly the case if they are in MIN. If $s = (\lambda x.u)\ v\ w_1 \cdots w_m$ then it is included if (a) all its reducts are terminating (which is satisfied if they are in MIN), and (b) the $\mathtt{head}\beta$-normal form s' of s has the right form, which holds because $s \sqsupset^+ s'$ (as $\Rightarrow_{\mathtt{head}\beta}$ is included in $\sqsupset$) and therefore $s' \in \mathsf{MIN}$ by assumption. □

In fact, we have that $\mathsf{MIN} \subseteq I \subseteq \mathsf{SN}$ for all RC-sets I. The latter inclusion is obvious by the termination requirement in the definition of RC-sets. The former inclusion follows easily:

Lemma 70. *For all RC-sets I, $\mathsf{MIN} \subseteq I$.*

Proof. We prove by induction on $\sqsupset$ that all elements of MIN are also in I. It is easy to see that if $s \in \mathsf{MIN}$ then s is neutral. Therefore, $s \in I$ if $t \in I$ whenever $s \sqsupset t$. But since MIN is closed by Lemma 69, each such t is in MIN, so also in I by the induction hypothesis. □

Aside from minimality of MIN, Lemma 70 actually provides I-computability of all variables, regardless of I. We prove this alongside termination of all I-computable terms.

Lemma 71. *Let I be an RC-set. The following statements hold for all types σ:*

1. *all variables $x : \sigma$ are I-computable*
2. *all I-computable terms $s : \sigma$ are terminating (w.r.t. $\sqsupset$)*

Proof. By a mutual induction on the form of σ, which we may safely write $\sigma_1 \to \ldots \to \sigma_m \to \iota$ (with $m \geq 0$ and $\iota \in \mathcal{S}$).

(1) By definition of I-computability, $x : \sigma$ is computable if and only if $x\ s_1 \cdots s_m \in I$ for all I-computable terms $s_1 : \sigma_1, \ldots, s_m : \sigma_m$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. However, as all σ_i are smaller types, we know that such terms s_i are terminating, so Lemma 70 gives the required result.

(2) Let $x_1 : \sigma_1, \ldots, x_m : \sigma_m$ be variables; by the induction hypothesis they are computable, and therefore $s\ x_1 \cdots x_m$ is in I and therefore terminating. Then the head, s, cannot itself be non-terminating (by monotonicity of $\sqsupset$). □

While SN is indisputably the easiest RC-set to define and work with, it will be beneficial for the strength of the method to consider a set strictly between MIN and SN. To this end, we assume given an ordering on types, and a function mapping each function symbol $\mathtt{f}$ to a set $Acc(\mathtt{f})$ of arguments positions. Here, we deviate from the text by not fixing Acc; again, this generality is not needed for the current paper, but is done with an eye on future extensions.

Definition 72 (Generalisation of Def. 11). *Assume given a quasi-ordering $\succeq^{\mathcal{S}}$ on $\mathcal{S}$ whose strict part $\succ^{\mathcal{S}} := \succeq^{\mathcal{S}} \setminus \preceq^{\mathcal{S}}$ is well-founded. Let $\approx^{\mathcal{S}}$ denote the corresponding equivalence relation $\approx^{\mathcal{S}} := \succeq^{\mathcal{S}} \cap \preceq^{\mathcal{S}}$.*

For a type $\sigma \equiv \sigma_1 \to \ldots \to \sigma_m \to \kappa$ (with $\kappa \in \mathcal{S}$) and sort ι, we write $\iota \succeq^{\mathcal{S}}_{+} \sigma$ if $\iota \succeq^{\mathcal{S}} \kappa$ and $\iota \succ^{\mathcal{S}}_{-} \sigma_i$ for each i, and we write $\iota \succ^{\mathcal{S}}_{-} \sigma$ if $\iota \succ^{\mathcal{S}} \kappa$ and $\iota \succeq^{\mathcal{S}}_{+} \sigma_i$ for each i.

For $\mathtt{f} : \sigma_1 \to \ldots \to \sigma_m \to \iota$ we assume given a set $Acc(\mathtt{f}) \subseteq \{i \mid 1 \leq i \leq m \wedge \iota \succeq^{\mathcal{S}}_{+} \sigma_i\}$. For $x : \sigma_1 \to \ldots \to \sigma_m \to \iota \in \mathcal{V}$, we write $Acc(x) = \{i \mid 1 \leq i \leq m \wedge \sigma_i = \tau_1 \to \ldots \to \tau_n \to \kappa \wedge \iota \succeq^{\mathcal{S}} \kappa\}$. 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 $i \in Acc(a)$ with $a \notin FV(s_i)$.

Remark: This definition of the accessibility relations deviates from, e.g., [9] by using a pair of relations ($\succeq^{\mathcal{S}}_{+}$ and $\succ^{\mathcal{S}}_{-}$) rather than positive and negative positions. This is not an important difference, but simply a matter of personal preference; using a pair of relations avoids the need to discuss type positions in the text, allowing for a shorter presentation. It is also not common to allow a choice in $Acc(\mathtt{f})$, but rather to fix $Acc(\mathtt{f}) = \{\sigma_i \mid 1 \leq i \leq m \wedge \iota \succeq^{\mathcal{S}}_{+} \sigma\}$ for *some* symbols (for instance constructors) and $Acc(\mathtt{f}) = \emptyset$ for the rest. We elected to leave the choice open for greater generality.

The interplay of the positive and negative relations $\succeq^{\mathcal{S}}_{+}$ and $\succ^{\mathcal{S}}_{-}$ leads to an important result on RC-sets.

Lemma 73. *Fix a sort $\iota \in \mathcal{S}$. Suppose I, J are RC-sets such that $I(\kappa) = J(\kappa)$ for all κ with $\iota \succ^{\mathcal{S}} \kappa$ and $I(\kappa) \subseteq J(\kappa)$ if $\iota \approx^{\mathcal{S}} \kappa$. Let $s : \sigma$. Then we have:*

- *If $\iota \succeq^{\mathcal{S}}_{+} \sigma$, then if s is I-computable also s is J-computable.*
- *If $\iota \succ^{\mathcal{S}}_{-} \sigma$, then if s is J-computable also s is I-computable.*

Proof. We prove both statements together by a shared induction on the form of σ. We can always write $\sigma \equiv \sigma_1 \to \ldots \to \sigma_m \to \kappa$ with $\kappa \in \mathcal{S}$.

First suppose $\iota \succeq^{\mathcal{S}}_{+} \sigma$; then $\iota \succeq^{\mathcal{S}} \kappa$ – so $I(\kappa) \subseteq J(\kappa)$ – and each $\iota \succ^{\mathcal{S}}_{-} \sigma_i$. Assume that s is I-computable; we must show that it is J-computable, so that for all J-computable $t_1 : \sigma_1, \ldots, t_m : \sigma_m$ we have: $s\ t_1 \cdots t_m \in J$. However, by the induction hypothesis each t_i is also I-computable, so $s\ t_1 \cdots t_m \in I(\kappa) \subseteq J(\kappa)$ by the assumption.

For the second statement, suppose $\iota \succ^{\mathcal{S}}_{-} \sigma$; then $\iota \succ^{\mathcal{S}} \kappa$, so $I(\kappa) = J(\kappa)$. Assume that s is J-computable; I-computability follows if $s\ t_1 \cdots t_m \in I(\kappa) = J(\kappa)$ whenever $t_1, \ldots, t_m$ are I-computable. By the induction hypothesis they are J-computable, so this holds by assumption. □

The RC-set C whose existence is asserted below offers computability with a notion of accessibility. It is worth noting that this is *not* a standard definition, but is designed to provide an additional relationship $\Rrightarrow_I$ that is terminating on computable terms. This relation will be useful in termination proofs using static DPs.

Theorem 74. *Let $\Rrightarrow_I$ be the relation on base-type terms where* $\mathtt{f}\ s_1 \cdots s_m \Rrightarrow_I s_i\ t_1 \cdots t_n$ *whenever* $i \in \mathit{Acc}(\mathtt{f})$ *and* $s_i : \sigma_1 \to \ldots \to \sigma_n \to \iota$ *and each* t_j *is* I*-computable.*

There exists an RC-set C *such that* $C = \{s \in \mathcal{T}(\mathcal{F}, \mathcal{V}) \mid s$ *has base type* $\wedge\ s$ *is terminating under* $\sqsupset \cup \Rrightarrow_C$ *and if* $s \sqsupset^* \mathtt{f}\ s_1 \cdots s_m$ *then* s_i *is* C*-computable for all* $i \in \mathit{Acc}(\mathtt{f})\}$.

Proof. We will define, by well-founded induction on ι using $\succeq^{\mathcal{S}}$, a set A_ι of terms as follows.

Assume A_κ has already been defined for all κ with $\iota \succ^{\mathcal{S}} \kappa$, and let X_ι be the set of RC-sets I such that $I(\kappa) = A_\kappa$ whenever $\iota \succ^{\mathcal{S}} \kappa$. We observe that X_ι is a complete lattice with respect to $\subseteq$: defining the bottom element $\sqcup\emptyset := \bigcup\{A_\kappa \mid \iota \succ^{\mathcal{S}} \kappa\} \cup \mathsf{MIN}$ and the top element $\sqcap\emptyset := \bigcup\{A_\kappa \mid \iota \succ^{\mathcal{S}} \kappa\} \cup \bigcup\{ \mathsf{SN}(\kappa) \mid \neg(\iota \succ^{\mathcal{S}} \kappa)\}$, and letting $\sqcup Z := \bigcup Z$, $\sqcap Z := \bigcap Z$ for non-empty Z, it is easily checked that $\sqcap$ and $\sqcup$ give a greatest lower and least upper bound within X_ι respectively. Now for an RC-set $I \in X_\iota$, we let:

$$\begin{aligned} F_\iota(I) = {} & \{s \in I \mid s : \kappa \not\approx^{\mathcal{S}} \iota\} \\ & \cup \{s \in \mathcal{T}(\mathcal{F},\mathcal{V}) \mid s : \kappa \approx^{\mathcal{S}} \iota \wedge s \text{ is terminating} \\ & \quad \text{under } \sqsupset \cup \Rrightarrow_I \wedge \text{ if } s \sqsupset^* \mathtt{f}\ s_1 \cdots s_m \text{ for a symbol} \\ & \quad \mathtt{f} \in \mathcal{F} \text{ then } \forall i \in \mathit{Acc}(\mathtt{f}) : s_i \text{ is } I\text{-computable}]\} \end{aligned}$$

Clearly, F_ι maps elements of X_ι to X_ι: terms of type $\kappa \not\approx^{\mathcal{S}} \iota$ are left alone, and $F_\iota(I)$ satisfies the properties to be an RC-set. Moreover, F_ι is monotone. To see this, let $I, J \in X_\iota$ such that $I \subseteq J$; we must see that $F_\iota(I) \subseteq F_\iota(J)$. To this end, let $s \in F_\iota(I)$; we will see that also $s \in F_\iota(J)$. This is immediate if $s : \kappa \not\approx^{\mathcal{S}} \iota$, as membership in X_ι guarantees that $F_\iota(I)(\kappa) = I(\kappa) \subseteq J(\kappa) = F_\iota(J)(\kappa)$. So assume $s : \kappa \approx^{\mathcal{S}} \iota$. We must see two things:

- s is terminating under $\sqsupset \cup \Rrightarrow_J$. We show that $\sqsupset \cup \Rrightarrow_J \subseteq \sqsupset \cup \Rrightarrow_I$; as s is terminating in the latter, the requirement follows. Clearly $\sqsupset \subseteq \sqsupset \cup \Rrightarrow_I$, so assume $s \Rrightarrow_J s'$. Then $s = \mathtt{f}\ t_1 \cdots t_m$ and $s' = t_i\ u_1 \cdots u_n$ for $i \in \mathit{Acc}(\mathtt{f})$ and J-computable $u_1, \ldots, u_n$. We can write $t_i : \sigma_1 \to \ldots \to \sigma_n \to \kappa$ and since $i \in \mathit{Acc}(\mathtt{f})$ we have $\iota \succeq^{\mathcal{S}} \kappa$ and $\iota \succ^{\mathcal{S}}_{-} \sigma_j$ for each j. By Lemma 73 then each u_j is also I-computable, so also $s \Rrightarrow_I s_1$.
- If $s \sqsupset^* \mathtt{f}\ s_1 \cdots s_m$ for some symbol $\mathtt{f}$ then for all $i \in \mathit{Acc}(\mathtt{f})$: s_i is J-computable. But this is obvious: as $s \in F_\iota(I)$, we know that such s_i are I-computable, and since $\iota \succeq^{\mathcal{S}}_{+} \sigma_i$ for $i \in \mathit{Acc}(\mathtt{f})$, Lemma 73 provides J-computability.

Thus, F is a monotone function on a complete lattice; by Tarski's fixpoint theorem there is a fixpoint, so an RC-set I such that for all sorts κ:

- if $\iota \succ^{\mathcal{S}} \kappa$ then $I(\kappa) = A_\kappa$;
- if $\iota \approx^{\mathcal{S}} \kappa$ then $I(\kappa) = \{s \in \mathcal{T}(\mathcal{F},\mathcal{V}) \mid s : \kappa \wedge s$ is terminating under $\sqsupset \cup \Rrightarrow_I \wedge$ if $s \sqsupset^* \mathtt{f}\ s_1 \cdots s_m$ for a symbol $\mathtt{f}$ then $\forall i \in \mathit{Acc}(\mathtt{f}) : s_i$ is I-computable$\}$

We define $A_\kappa := I(\kappa)$ for all $\kappa \approx^{\mathcal{S}} \iota$.

Now we let $C := \bigcup_{\iota \in \mathcal{S}} A_\iota$. Clearly, C satisfies the given requirement. □

Thm. 74 easily gives the proof of Thm. 13 in the text:

Proof (Proof of Thm. 13). Thm. 13 follows by taking $\Rrightarrow_{\mathcal{R}}$ for $\sqsupset$ (which satisfies the requirements by Lemma 68) and taking for each $Acc(\mathtt{f})$ the maximum set $\{i \mid 1 \leq i \leq m \wedge \iota \succeq^{\mathcal{S}}_{+} \sigma_i\}$. $\square$

A.2 Additional properties of computable terms

For reasoning about computable terms (as we will do when defining static DPs and reasoning about computable chains), there are a number of properties besides those in Lemma 71 that will prove very useful to have. In the following, we fix the RC-set C as obtained from Thm. 74.

Lemma 75. *If s is C-computable and $s \sqsupset t$, then t is also C-computable.*

(This actually holds for any RC-set, but we will only use it for C.)

Proof. By induction on the type of s. If s has base type, then C-computability implies that $s \in C$, and because C is an RC-set all reducts of s are also in C. Otherwise, $s : \sigma \rightarrow \tau$ and computability of s implies computability of $s\ u$ for all computable $u : \sigma$. By the induction hypothesis, the fact that $s\ u \sqsupset t\ u$ by monotonicity of $\sqsupset$ implies that $t\ u$ is computable for all computable u, and therefore by definition t is computable. $\square$

Thus, computability is preserved under $\sqsupset$; the following result shows that it is also preserved under $\Rrightarrow_C$.

Lemma 76. *If s is C-computable and $s \Rrightarrow_C t$, then t is also C-computable.*

Proof. If $s \Rrightarrow_C t$, then both terms have base type, so C-computability is simply membership in C. We have $s = \mathtt{f}\ s_1 \cdots s_m$ and $t = s_i\ t_1 \cdots t_n$ with each t_j C-computable. Since, by definition of C, also s_i is C-computable, C-computability of t immediately follows. $\square$

Finally, we will see that C-computability is also preserved under $\trianglerighteq_{\mathtt{acc}}$. For this, we first make a more general statement, which will also handle variables below binders (which are freed in subterms).

Lemma 77. *Let $s : \sigma_1 \rightarrow \ldots \rightarrow \sigma_m \rightarrow \iota$ and $t : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \kappa$ be meta-terms, such that $s \trianglerighteq_{\mathtt{acc}} t$. Let γ be a substitution with $FMV(s) \subseteq \mathtt{dom}(\gamma) \subseteq \mathcal{M}$.*

Let $u_1 : \tau_1, \ldots, u_n : \tau_n$ be C-computable terms, and δ a substitution with $\mathtt{dom}(\delta) \subseteq \mathcal{V}$ such that each $\delta(x)$ is C-computable, and for $t' := (t(\gamma \cup \delta))\ u_1 \cdots u_n$ there is no overlap between $FV(t')$ and the variables bound in s.

Then there exist a C-computable substitution ξ with $\mathtt{dom}(\xi) \subseteq \mathcal{V}$ and C-computable terms $v_1 : \sigma_1, \ldots, v_m : \sigma_m$ such that we have $(s(\gamma \cup \xi))\ v_1 \cdots v_m\ (\Rrightarrow_C \cup \Rightarrow_{\mathtt{head}\beta})^*\ t'$.

Proof. We prove the lemma by induction on the derivation of $s \trianglerighteq_{\mathtt{acc}} t$; in this, we can assume (by α-conversion) that variables that occur bound in s do not also occur free or occur in γ.

If $s = t$, then we are done choosing ξ and $\boldsymbol{v}$ equal to δ and $\boldsymbol{u}$.

If $s = \lambda x.s'$ with $x : \sigma_1$ and $s' \trianglerighteq_{\mathtt{acc}} t$, then by the assumption based on α-conversion above, x does not occur free in s or in the range of γ. By the induction hypothesis, there exist a computable substitution ξ' and computable terms $v_2, \dots, v_m$ such that $(s'(\gamma \cup \xi'))\ v_2 \cdots v_m\ (\Rrightarrow_C \cup \Rightarrow_{\mathtt{head}\beta})^*\ t'$. We can safely assume that x does not occur in the range of ξ', since x does not occur in t' either (if x does occur, we can replace it by a different variable). Therefore, if we define $\xi := [x := x] \cup [y := \xi'(y) \mid y \in \mathcal{V} \wedge y \neq x]$, we have $s'(\gamma \cup \xi') = (s'(\gamma \cup \xi))[x := \xi'(x)]$. Choosing $v_1 := \xi'(x)$, we get $(s(\gamma \cup \xi))\ v_1 \cdots v_m \Rightarrow_{\mathtt{head}\beta} (s'(\gamma \cup \xi'))\ v_2 \cdots v_m\ (\Rrightarrow_C \cup \Rightarrow_{\mathtt{head}\beta})^*\ t'$.

If $s = x\ s_1 \cdots s_j$ for $s_i : \pi_1 \to \dots \to \pi_{n'} \to \kappa'$ with $\iota \succeq^{\mathcal{S}} \kappa'$ and $x \notin FV(s_i)$ and $s_i \trianglerighteq_{\mathtt{acc}} t$, then the induction hypothesis provides C-computable terms $w_1 : \pi_1, \dots, w_{n'} : \pi_{n'}$ and a substitution ξ' such that $(s_i(\gamma \cup \xi'))\ w_1 \cdots w_{n'}\ (\Rrightarrow_C \cup \Rightarrow_{\mathtt{head}\beta})^*\ t'$. Since $x \notin FV(s_i)$ by definition of $\trianglerighteq_{\mathtt{acc}}$, we can safely assume that $x \notin \mathtt{dom}(\xi')$. Now recall that by assumption $\mathcal{F}$ contains infinitely many constructors of all types; let $\mathtt{c} : \kappa \to \iota$ be a symbol that does not occur anywhere in $\mathcal{R}$. We can safely assume that $Acc(\mathtt{c}) = \{1\}$. Then $w := \lambda y_1 \dots y_j z_1 \dots z_m.\mathtt{c}\ (y_i\ w_1 \cdots w_{n'})$ is C-computable. Now let $\xi := [x := w] \cup [y := \xi'(y) \mid y \in \mathcal{V} \wedge y \neq x]$, and let $v_1, \dots, v_m$ be variables (which are C-computable by Lemma 71(1)). Then $(s(\gamma \cup \xi))\ v_1 \cdots v_m \Rightarrow_{\mathtt{head}\beta}^{j+m} s_i(\gamma \cup \xi')\ w_1 \cdots w_n'\ (\Rrightarrow_C \cup \Rightarrow_{\mathtt{head}\beta})^*\ t'$.

Otherwise, $s = \mathtt{f}\ s_1 \cdots s_n$ and $s_i \trianglerighteq_{\mathtt{acc}} t$ for some $i \in Acc(\mathtt{f})$; by the induction hypothesis there exist ξ and C-computable terms $w_1, \dots, w_{n'}$ such that $s' := (s_i(\gamma \cup \xi))\ w_1 \cdots w_{n'}\ (\Rrightarrow_C \cup \Rightarrow_{\mathtt{head}\beta})^* t'$. We have $(s(\gamma \cup \xi))\ v_1 \cdots v_m \Rrightarrow_C s'$ for any $\boldsymbol{v}$ (e.g., variables). □

From this we conclude:

Lemma 78. *Let s be a closed meta-term, γ a substitution with $FMV(s) \subseteq \mathtt{dom}(\gamma) \subseteq \mathcal{M}$ and t such that $s \trianglerighteq_{\mathtt{acc}} t$ and $s\gamma$ is C-computable. Then for all substitutions δ mapping $FV(t)$ to computable terms: $t(\gamma \cup \delta)$ is C-computable.*

Proof. $t(\gamma \cup \delta)$ is C-computable if $(t(\gamma \cup \delta))\ u_1 \cdots u_n$ is C-computable for all computable $u_1, \dots, u_n$. By Lemma 77 and the fact that s is closed, there exist C-computable terms $v_1, \dots, v_m$ such that $(s\gamma)\ v_1 \cdots v_m\ (\Rrightarrow_C \cup \Rightarrow_{\mathtt{head}\beta})^*\ (t(\gamma \cup \delta))\ u_1 \cdots u_n$. But $s\gamma$ is C-computable, and therefore so is $(s\gamma)\ v_1 \cdots v_m$. Since $\Rrightarrow_C$ and $\Rightarrow_{\mathtt{head}\beta}$ are both computability-preserving by Lemmas 76 and 75 respectively (as $\Rightarrow_{\mathtt{head}\beta}$ is included in $\sqsupset$) we are done. □

Lemma 79. *A neutral term is C-computable if and only if all its $\sqsupset$-reducts are C-computable.*

Proof. That C-computability of a term implies C-computability of its reducts is given by Lemma 75. For the other direction, let $s : \sigma_1 \to \dots \to \sigma_m \to \iota$ be neutral and suppose that all its reducts are C-computable. To prove that also s is C-computable, we must see that for all C-computable terms $t_1 : \sigma_1, \dots, t_m : \sigma_m$ the term $u := s\ t_1 \cdots t_m$ is in C. We prove this by induction on $(t_1, \dots, t_m)$ ordered by $\sqsupset_{\mathtt{prod}}$. Clearly, since s does not have the form $\mathtt{f}\ s_1 \cdots s_n$ with $Acc(\mathtt{f}) \neq \emptyset$, nor does u, so $u \in C$ if all its reducts are in C. But since s is neutral, all reducts of u either have the form $s'\ t_1 \cdots t_m$ with $s \sqsupset s'$ – which is in C because all t_i are C-computable and s' is computable as a reduct of s – or the form $s\ t_1 \cdots t_i' \cdots t_m$ with $t_i \sqsupset t_i'$ – which is in C by the induction hypothesis. □

Using the $\Rightarrow_{\mathtt{head}\beta}$-restrictions on $\sqsupset$, we obtain the following result:

Lemma 80. *Let $x : \sigma \in \mathcal{V}$. A term $\lambda x.s \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ is C-computable if and only if $s[x := t]$ is computable for all C-computable $t : \sigma$.*

Proof. If $\lambda x.s$ is C-computable, then by definition so is $(\lambda x.s)\ t$ for all C-computable t; by Lemma 75 and inclusion of $\Rightarrow_{\mathtt{head}\beta}$ in $\sqsupset$, this implies C-computability of the reducts $s[x := t]$.

For the other direction, suppose $s[x := t]$ is C-computable for all C-computable $t : \sigma$. To obtain C-computability of $\lambda x.s$, we must see that $(\lambda x.s)\ t$ is C-computable for all C-computable $t : \sigma$. As $(\lambda x.s)\ t$ is neutral, this holds if all its $\sqsupset$-reducts u are C-computable by Lemma 79, and certainly if all its $\sqsupset^+$-reducts are C-computable, which we prove by induction on u oriented with $\sqsupset$. But by definition of $\sqsupset$ (and induction on the derivation $(\lambda x.s)\ t \sqsupset^+ u$) there exists a term v such that $s[x := t] \sqsupset^* v$ and $u \Rightarrow^*_{\mathtt{head}\beta} v$. If $u = v$, we thus obtain the required property, and if $u \Rightarrow^+_{\mathtt{head}\beta} v$, then u is neutral and hence is C-computable if all its $\sqsupset$-reducts are, which is the case by the induction hypothesis. □

B Static dependency pairs

In this appendix, we will first prove the main result from § 4: Thm. 37. Then, we will prove the "inverse" result, Thm. 34. Finally, to provide a greater context to the current work, we will discuss how the definitions in [34,46] relate to the definitions here.

B.1 Static dependency pairs: the main result

To start, we prove Thm. 37, which states that a properly applied, accessible function passing AFSM with rules $\mathcal{R}$ is terminating if it admits no $\mathcal{R}$-computable formative $(SDP(\mathcal{R}), \mathcal{R})$-dependency chains. Thm. 32, which states that an AFSM is terminating if it admits no $(SDP(\mathcal{R}), \mathcal{R})$-dependency chains, follows as a corollary.

In the following, let $C = C_{\mathcal{R}}$ be a computability predicate following Def. 74 for $\sqsupset$ the rewrite relation $\Rightarrow_{\mathcal{R}}$. We will briefly call a term "computable" if it is C-computable.

Henceforth, we will assume without explicitly stating it that $(\mathcal{F}, \mathcal{R})$ is properly applied, so we can speak of $minar(\mathtt{f})$ without disclaimers; we let $minar(\mathtt{f}) = 0$ for $\mathtt{f} \notin \mathcal{D}$. We start with an observation on the consequences of accessible function passingness:

Lemma 81. *Let ℓ be a closed pattern, Z a meta-variable and $x_1, \dots, x_k$ variables such that $\ell \trianglerighteq_{\mathtt{acc}} Z\langle x_1, \dots, x_k\rangle$. If $\ell\gamma$ is a computable term, then so is $\gamma(Z)$.*

Proof. Since ℓ is closed, $\ell(\gamma \cup \delta) = \ell\gamma$ is computable for all computable substitutions δ whose domain is contained in $\mathcal{V}$. By Lemma 78, we thus have computability of $Z\langle x_1, \dots, x_k\rangle(\gamma \cup \delta)$ for all such δ. Since $\ell\gamma$ is a term, $Z \in \mathtt{dom}(\gamma)$ so we can either write $\gamma(Z) = \lambda x_1 \dots x_k.s$ or $\gamma(Z) = \lambda x_1 \dots x_i.s'$ with $i < k$ and s' not an abstraction.

In the first case, if we let $\delta := [x_1 := u_1, \dots, x_k := u_k]$ for computable terms $u_1, \dots, u_k$ we have computability of $Z\langle x_1, \dots, x_k\rangle(\gamma \cup \delta) = s[x_1 := u_1, \dots, x_k := u_k]$. Since this holds for *all* computable $u_1, \dots, u_k$, Lemma 80 provides computability of $\lambda x_1 \dots x_k.s = \gamma(Z)$. In the second case, the same substitution δ provides computability of $s'[x_1 := u_1, \dots, x_i := u_i]\ u_{i+1} \cdots u_n$ which (since this holds for *any* $u_{i+1}, \dots, u_n$) implies computability of $s'[x_1 := u_1, \dots, x_i := u_i]$, and this in turn implies computability of $\gamma(Z)$ as before. □

Thus, computability of the left-hand side of an instantiated DP implies computability of all instantiated meta-variables. To transfer this property to the right-hand side of the instantiated pair, we have a closer look at the relation $\trianglerighteq_A$.

In the following, we say that a meta-term s *respects minar* if $s \trianglerighteq_\beta \mathtt{f}\ t_1 \cdots t_n$ implies $n \geq \mathit{minar}(\mathtt{f})$.

Lemma 82. *Let s be a meta-term that respects minar and γ a substitution on a finite domain with $FMV(s) \subseteq \mathtt{dom}(\gamma) \subseteq \mathcal{M}$, such that all $\gamma(Z)$ are computable. If there exists a computable substitution δ on a variable domain (that is, $\mathtt{dom}(\gamma) \subseteq \mathcal{V}$) such that $s(\gamma \cup \delta)$ is not computable, then there exists a pair $t\ (A) \in \mathsf{cand}(s)$ such that all of the following hold:*

- *there is a computable substitution δ on variable domain such that $t(\gamma \cup \delta)$ is not computable;*
- *γ respects A;*
- *for all $t' \neq t$ such that $t \trianglerighteq_B t'$ holds for some B respected by γ: $t'(\gamma \cup \delta)$ is computable for all computable substitutions δ on variable domain.*

Proof. Let S be the set of all pairs $t\ (A)$ such that (a) $s \trianglerighteq_A t$, (b) there exists a computable substitution δ on variable domain such that $t(\gamma \cup \delta)$ is not computable, and (c) γ respects A. This set is non-empty, as it contains $\{s\ (\emptyset)\}$. As the relations $\trianglerighteq_\beta$ and $\supseteq$ are both well-founded quasi-orderings (the latter on finite sets), we can select a pair $t\ (A)$ that is, in a sense, *minimal* in S: for all $t'\ (A') \in S$: if $t \trianglerighteq_\beta t'$ then $t = t'$ and not $A' \subsetneq A$ (it is possible that A and A' are incomparable). We observe that for all t', B such that $t' \neq t$ and $t \trianglerighteq_B t'$ and γ respects B we cannot have $t'\ (A \cup B) \in S$ by minimality of $t\ (A)$, so since $s \trianglerighteq_{A \cup B} t \trianglerighteq_{A \cup B} t'$ and clearly γ respects $A \cup B$, it can only follow that requirement (b) is not satisfied for t'.

Now suppose that t has the form $\mathtt{f}\ t_1 \cdots t_n$. Then by the above reasoning, all $t_i(\gamma \cup \delta)$ are computable, and by definition of "s respects *minar*" we know that $n \geq k := \mathit{minar}(\mathtt{f})$. Thus, $(\mathtt{f}\ t_1 \cdots t_k)(\gamma \cup \delta)$ is not computable (since otherwise $t(\gamma \cup \delta)$ would be computable), and by definition of S as a set of BRSMTs of s

(and minimality of A) we have $\mathtt{f}\ t_1 \cdots t_k\ (A) \in \mathsf{cand}(s)$. By minimality of t in S, we see that $\mathtt{f}\ t_1 \cdots t_k\ (A)$ satisfies all the requirements for the lemma to hold.

Thus, if t has the form $\mathtt{f}\ t_1 \cdots t_n$, we are done; towards a contradiction we will show that if t does *not* have this form, then $t\ (A)$ is not minimal.

Consider the form of t:

– $t = \lambda x.t'$: by Lemma 80, non-computability of $t(\gamma \cup \delta)$ implies non-computability of $t'(\gamma \cup \delta)[x := u]$ for some computable u. Since, by α-conversion, we can assume that x does not occur in domain or range of γ or δ, we have non-computability of $t'(\gamma \cup \delta \cup [x := u])$, and $\delta \cup [x := u]$ is a computable substitution on variable domain while $t \trianglerighteq_A t'$.
– $t = x\ t_1 \cdots t_n$ with $x \in \mathcal{V}$: whether $x \in \mathtt{dom}(\delta)$ or not, $\delta(x)$ is computable (either by the assumption on δ or by Lemma 71(1)). Therefore, the only way for $t(\gamma \cup \delta)$ to not be computable is if some $s_i(\gamma \cup \delta)$ is not computable, and $s \trianglerighteq_A s_i$.
– $t = \mathtt{c}\ t_1 \cdots t_n$ with $\mathtt{c} \in \mathcal{F} \setminus \mathcal{D}$: $t(\gamma \cup \delta)$ is non-computable only if there exist computable terms $u_{n+1}, \ldots, u_m$ such that the term $\mathtt{c}\ t_1 \cdots t_n\ u_{n+1} \cdots u_m$ of base type is not in C. This can only be the case if it is non-terminating or some $t_i(\gamma \cup \delta)$ is not computable. Since head-reductions are impossible, non-termination implies non-termination of some $t_i(\gamma \cup \delta)$ or u_j, which by Lemma 71(2) implies non-computability; as all u_j are computable by assumption, this means some $t_i(\gamma \cup \delta)$ is non-computable. We are done because $t \trianglerighteq_A t_i$.
– $t = \mathtt{f}\ t_1 \cdots t_n$ with $\mathtt{f} \in \mathcal{D}$ but $n < \mathit{arity}(\mathtt{f})$: same as above, because terms of this form cannot be reduced at the head (or the root).
– $t = (\lambda x.u)\ t_0 \cdots t_n$: $t(\gamma \cup \delta)$ is neutral, so by Lemma 79 non-computability implies the non-computability of a reduct. If the reduct $u(\gamma \cup \delta)[x := t_0(\gamma \cup \delta)]\ (t_1(\gamma \cup \delta)) \cdots (t_n(\gamma \cup \delta)) = (u[x := t_0]\ t_1 \cdots t_n)(\gamma \cup \delta)$ is non-computable, we are done because $t \trianglerighteq_A u[x := t_0]\ t_1 \cdots t_n$. Otherwise, note that all many-step reducts of $t(\gamma \cup \delta)$ are either also a reduct of $(u[x := t_0]\ t_1 \cdots t_n)(\gamma \cup \delta)$ – and therefore computable – or have the form $(\lambda x.u')\ t'_0 \cdots t'_n$ with $u(\gamma \cup \delta) \Rightarrow^*_{\mathcal{R}} u'$ and each $t_i(\gamma \cup \delta) \Rightarrow^*_{\mathcal{R}} t'_i$. Thus, at least one of $u(\gamma \cup \delta)$ or $t_i(\gamma \cup \delta)$ has to be non-terminating. But if $u(\gamma \cup \delta)$ is non-terminating, then so is $u[x := u'](\gamma \cup \delta)$, contradicting computability of $(u[x := t_0]\ t_1 \cdots t_n)(\gamma \cup \delta)$. The same holds if $t_i(\gamma \cup \delta)$ is non-terminating for some $i \geq 1$. Thus, $t_0(\gamma \cup \delta)$ is non-terminating and therefore non-computable, and we indeed have $t \trianglerighteq_A t_0$.
– $t = Z\langle s_1, \ldots, s_k\rangle\ t_1 \cdots t_n$: we either have $\gamma(Z) = \lambda x_1 \ldots x_k.u$ or $\gamma(Z) = \lambda x_1 \ldots x_i.u'$ with $i < k$ and u' not an abstraction; in the latter case let $u := \lambda x_{i+1} \ldots x_k.u'\ x_{i+1} \cdots x_k$. Either way, $t(\gamma \cup \delta) = u[x_1 := s_1(\gamma \cup \delta), \ldots, x_k := s_k(\gamma \cup \delta)]\ (t_1(\gamma \cup \delta)) \cdots (t_n(\gamma \cup \delta))$.
For this term to be non-computable, either some $t_i(\gamma \cup \delta)$ should be non-computable, or $u[x_1 := s_1(\gamma \cup \delta), \ldots, x_k := s_k(\gamma \cup \delta)]$. The former case immediately contradicts minimality, since $t \trianglerighteq_\emptyset t_i$, so we assume the latter. However, if all $s_i(\gamma \cup \delta)$ are computable, then so is $u[x_1 := s_1(\gamma \cup \delta), \ldots, x_k := s_k(\gamma \cup \delta)]$:
  - if $\gamma(Z) = \lambda x_1 \ldots x_k.u$ then this holds by computability of all $\gamma(Z)$ and Lemma 80;

- if $\gamma(Z) = \lambda x_1 \dots x_i.u'$ with $i < k$ and $u = u'\ x_{i+1} \cdots x_k$, then computability of $\gamma(Z)$ and Lemma 80 provide computability of $u'[x_1 := q_1, \dots, x_i := q_i]$, which by definition of computability for higher-order terms implies computability for $u'[x_1 := q_1, \dots, x_i := q_i]\ q_{i+1} \cdots q_n = u[x_1 := q_1, \dots, x_n := q_n]$.

Thus, some $s_i(\gamma \cup \delta)$ must be non-computable, and since substituting an unused variable has no effect, this must be the case for some i with $x_i \in FV(u)$. So in this case, γ respects $B := A \cup \{Z : i\}$ and we obtain $s \trianglerighteq_B t \trianglerighteq_B s_i$. $\square$

Next, let us consider formative reductions. We will prove that reductions from a terminating term to some instance of a pattern may be assumed to be formative.

Lemma 83. *Let ℓ be a pattern and γ a substitution on domain $FMV(\ell)$ such that a meta-variable Z with $arity(Z) = k$ is mapped to a term $\lambda x_1 \dots x_k.t$. Let s be a terminating term. If $s \Rightarrow_{\mathcal{R}}^* \ell\gamma$, then there exists a substitution δ on the same domain as γ such that each $\delta(Z) \Rightarrow_{\mathcal{R}}^* \gamma(Z)$ and $s \Rightarrow_{\mathcal{R}}^* \ell\delta$ by an ℓ-formative reduction.*

Note that the restriction on γ is very light: every substitution γ on domain $FMV(\ell)$ can be altered to map meta-variables with arity k to terms with k abstracted variables: if $\gamma(Z) = \lambda x_1 \dots x_k.t$ with $k = arity(Z)$ then let $\gamma'(Z) = \gamma(Z)$, and if $\gamma(Z) = \lambda x_1 \dots x_i.t$ with $i < arity(Z)$ and t not an abstraction, then replace this by setting $\gamma'(Z) := \lambda x_1 \dots x_k.t\ x_{i+1} \cdots x_k$. Note that $Z\langle x_1, \dots, x_k\rangle\gamma = Z\langle x_1, \dots, x_k\rangle\gamma'$. Therefore, we always have $\ell\gamma = \ell\gamma'$.

Proof. We prove the lemma by induction first on s ordered by $\Rightarrow_{\mathcal{R}} \cup \triangleright$, second on the length of the reduction $s \Rightarrow_{\mathcal{R}}^* \ell\gamma$. If ℓ is not a fully extended linear pattern, then we are done choosing $\delta := \gamma$. Otherwise, we consider four cases:

1. ℓ is a meta-variable application $Z\langle x_1, \dots, x_k\rangle$;
2. ℓ is not a meta-variable application, and the reduction $s \Rightarrow_{\mathcal{R}}^* \ell\gamma$ does not contain any headmost steps;
3. ℓ is not a meta-variable application, and the reduction $s \Rightarrow_{\mathcal{R}}^* \ell\gamma$ contains headmost steps, the first of which is a $\Rightarrow_\beta$ step;
4. ℓ is not a meta-variable application, and the reduction $s \Rightarrow_{\mathcal{R}}^* \ell\gamma$ contains headmost steps, the first of which is not a $\Rightarrow_\beta$ step.

In the first case, if ℓ is a meta-variable application $Z\langle x_1, \dots, x_k\rangle$, then by α-conversion we may write $\gamma = [Z := \lambda x_1 \dots x_k.t]$ with $\ell\gamma = t$. Let δ be $[Z := \lambda x_1 \dots x_k.s]$. Then δ has the same domain as γ, and indeed $\delta(Z) = \lambda x_1 \dots x_k.s \Rightarrow_{\mathcal{R}}^* \lambda x_1 \dots x_k.(\ell\gamma) = \gamma(Z)$.

In the second case, a reduction without any headmost steps, note that s has the same outer shape as ℓ: either (a) $s = \lambda x.s'$ and $\ell = \lambda x.\ell'$, or (b) $s = a\ s_1 \cdots s_n$ and $\ell = a\ \ell_1 \cdots \ell_n$ for some $a \in \mathcal{V} \cup \mathcal{F}$ (since ℓ is a pattern, a cannot be a meta-variable application or abstraction if $n > 0$). In case (a), we obtain δ such that $s' \Rightarrow_{\mathcal{R}}^* \ell'\delta$ by an ℓ'-formative reduction and $\delta \Rightarrow_{\mathcal{R}}^* \gamma$ by the induction hypothesis (as sub-meta-terms of linear patterns are still linear patterns). In case (b), we let

γ_i be the restriction of γ to $FMV(\ell_i)$ for $1 \leq i \leq n$; by linearity of ℓ, all γ_i have non-overlapping domains and $\gamma = \gamma_1 \cup \cdots \cup \gamma_n$. The induction hypothesis provides $\delta_1, \dots, \delta_n$ on the same domains such that each $s_i \Rightarrow_{\mathcal{R}}^* \ell_i\delta_i$ by an ℓ_i-formative reduction and $\delta_i \Rightarrow_{\mathcal{R}}^* \gamma_i$; we are done choosing $\delta := \delta_1 \cup \cdots \cup \delta_n$.

In the third case, if the first headmost step is a β-step, note that s must have the form $(\lambda x.t)\ u\ q_1 \cdots q_n$, and moreover $s \Rightarrow_{\mathcal{R}}^* (\lambda x.t')\ u'\ q_1' \cdots q_n' \Rightarrow_\beta t'[x := u']\ q_1' \cdots q_n' \Rightarrow_{\mathcal{R}}^* \ell\gamma$ by steps in the respective subterms. But then also $s \Rightarrow_\beta t[x := u]\ q_1 \cdots q_n \Rightarrow_{\mathcal{R}}^* t'[x := u']\ q_1' \cdots q_n' \Rightarrow_{\mathcal{R}}^* \ell\gamma$, and we can get δ and an ℓ-formative reduction for $t[x := u]\ q_1 \cdots q_n \Rightarrow_{\mathcal{R}}^* \ell\delta$ by the induction hypothesis.

In the last case, if the first headmost step is not a β-step, then we can write $s = \mathtt{f}\ s_1 \cdots s_n \Rightarrow_{\mathcal{R}}^* \mathtt{f}\ s_1' \cdots s_n' = (\ell'\eta)\ s_{i+1}' \cdots s_n' \Rightarrow_{\mathcal{R}} (r\eta)\ s_{i+1}' \cdots s_n' \Rightarrow_{\mathcal{R}}^* \ell\gamma$ for some $\mathtt{f} \in \mathcal{D}$, terms $s_j \Rightarrow_{\mathcal{R}}^* s_j'$ for $1 \leq j \leq n$, rule $\ell' \Rightarrow r$ and substitution η on domain $FMV(\ell')$. But then $\ell'\ Z_{i+1} \cdots Z_n \Rightarrow r\ Z_{i+1} \cdots Z_n \in \mathcal{R}^{\mathsf{ext}}$, and for $\eta' := \eta \cup [Z_{i+1} := s_{i+1}', \dots, Z_n := s_n']$ we both have $s \Rightarrow_{\mathcal{R}}^* (\ell'\ Z_{i+1} \cdots Z_n)\eta'$ without any headmost steps, and $(r\ Z_{i+1} \cdots Z_n)\eta' \Rightarrow_{\mathcal{R}}^* \ell\gamma$. By the second induction hypothesis, there exists a substitution ξ such that $s \Rightarrow_{\mathcal{R}}^* (\ell'\ Z_{i+1} \cdots Z_n)\xi$ by a $(\ell'\ Z_{i+1} \cdots Z_n)$-formative reduction and $\xi \Rightarrow_{\mathcal{R}}^* \eta'$. This gives $s \Rightarrow_{\mathcal{R}}^+ (r\ Z_{i+1} \cdots Z_n)\xi \Rightarrow_{\mathcal{R}}^* (r\ Z_{i+1} \cdots Z_n)\eta' \Rightarrow_{\mathcal{R}}^* \ell\gamma$, so by the first induction hypothesis we obtain δ such that $(r\ Z_{i+1} \cdots Z_n)\xi \Rightarrow_{\mathcal{R}}^* \ell\delta$ by an ℓ-formative reduction, and $\delta \Rightarrow_{\mathcal{R}}^* \gamma$. $\square$

Essentially, Lemma 83 states that we can postpone reductions that are not needed to obtain an instance of the given pattern. This is not overly surprising, but will help us eliminate some proof obligations later in the termination proof.

From this, we have the main result on static dependency chains.

Theorem 37. *Let $(\mathcal{F}, \mathcal{R})$ be a* properly applied, accessible function passing *AFSM. If $(\mathcal{F}, \mathcal{R})$ is non-terminating, then there is an infinite $\mathcal{R}$-computable formative $(SDP(\mathcal{R}), \mathcal{R})$-dependency chain.*

Proof. In the following, let a *minimal non-computable term* be a term $s := \mathtt{f}\ s_1 \cdots s_k$ with $k = minar(\mathtt{f})$, such that $\mathtt{f} \in \mathcal{D}$, and s is not computable but all s_i are C-computable. We say that s is MNC.

We first observe that if $\Rightarrow_{\mathcal{R}}$ is non-terminating, then there exists a MNC term. After all, if $\Rightarrow_{\mathcal{R}}$ is non-terminating, then there is a non-terminating term s, which (by Lemma 71(2)) is also non-computable. Let t (A) be the element of $\mathsf{cand}(s)$ that is given by Lemma 82 for $\gamma = \delta = []$. Then $A = \emptyset$ and t has the form $\mathtt{f}\ t_1 \cdots t_k$ with $k = minar(\mathtt{f})$, and there exists a computable substitution δ such that $t\delta$ is not computable but all $t_i\delta$ are.

Thus, assuming $\Rightarrow_{\mathcal{R}}$ is non-terminating, we can select a MNC term t_{-1}.

Now for $i \in \mathbb{N}$, let a MNC term $t_{i-1} = \mathtt{f}\ q_1 \cdots q_k$ be given. By definition of computability, there are computable $q_{k+1}, \dots, q_m$ such that $\mathtt{f}\ q_1 \cdots q_k$ has base type and is not computable. Since all q_i are computable this implies that $\mathtt{f}\ q_1 \cdots q_k$ is non-terminating, and since they are terminating, there is eventually a reduction at the head: there exist a rule $\mathtt{f}\ \ell_1 \cdots \ell_k \Rightarrow r$ and a substitution γ such that $\mathtt{f}\ q_1 \cdots q_m \Rightarrow_{\mathcal{R},in}^* \mathtt{f}\ q_1' \cdots q_m'$ (where $\Rightarrow_{\mathcal{R},in}^*$ indicates a reduction in the argument terms q_j) $= \mathtt{f}\ (\ell_1\gamma) \cdots (\ell_k\gamma)\ q_{k+1}' \cdots q_m' \Rightarrow_{\mathcal{R}} (r\gamma)\ q_{k+1}' \cdots q_m'$, which latter term is still non-terminating. But then also $(r\gamma)\ q_{k+1} \cdots q_m$ is non-terminating (as it reduces to the term above), so $r\gamma$ is not computable.

From the above we conclude: $t_{i-1} \Rightarrow^*_{\mathcal{R},in} (\mathtt{f}\ \ell_1 \cdots \ell_k)\gamma \Rightarrow_{\mathcal{R}} r\gamma$, and $r\gamma$ is not computable. By Lemma 83, we can safely assume that the reductions $q_j \Rightarrow^*_{\mathcal{R}} \ell_j\gamma$ are ℓ_j-formative if $\mathtt{f}\ \ell_1 \cdots \ell_k$ is a fully extended linear pattern; and since $\mathtt{f}\ \ell_1 \cdots \ell_k$ is closed we can safely assume that $\mathtt{dom}(\gamma) = \mathit{FMV}(\mathtt{f}\ \ell_1 \cdots \ell_k)$.

Let $s_i := \mathtt{f}^\sharp\ (\ell_1\gamma) \cdots (\ell_k\gamma)$, and note that all $\ell_j\gamma$ are computable by Lemma 75. We observe that for all Z occurring in r we have that $\gamma(Z)$ is C-computable by a combination of accessible function passingness, computability of $\ell_j\gamma$ and Lemma 81. As $r\gamma$ is non-computable, Lemma 82 provides an element $t\ (A)$ of $\mathsf{cand}(r)$ with pleasant minimality properties and a computable substitution δ on domain $FV(t)$ such that γ respects A and $t(\gamma \cup \delta)$ is not computable. For $FV(t) = \{x_1, \ldots, x_n\}$, let $Z_1, \ldots, Z_n$ be fresh meta-variables; then $p := t[x_1 := Z_1, \ldots, x_n := Z_n] = \mathit{metafy}(t)$, and $p\eta = t\delta$ for η the substitution mapping $X \in \mathit{FMV}(\ell)$ to $\gamma(\ell)$ and each Z_j to $\delta(x_j)$.

Set $\rho_i := \ell^\sharp \Rrightarrow p^\sharp\ (A)$ and $t_i := p^\sharp\eta$. Then t_i is MNC, because the meta-term t supplied by Lemma 82 has the form $\mathtt{g}\ u_1 \cdots u_m$ with $m = \mathit{arity}(\mathtt{g})$ and $u_j(\gamma \cup \delta)$ is C-computable for each j because $t \trianglerighteq_A u_j$. Thus, we can continue the construction.

The chain $[(\rho_0, s_0, t_0), (\rho_1, s_1, t_1), \ldots]$ thus constructed is an infinite formative $(\mathit{SDP}(\mathcal{R}), \mathcal{R})$-dependency chain. That it is a $(\mathit{SDP}(\mathcal{R}), \mathcal{R})$-dependency chain is obvious because each $\rho_i \in \mathit{SDP}(\mathcal{R})$ (since $t\ (A) \in \mathsf{cand}(r)$), because γ respects A and η corresponds with γ on all meta-variables that take arguments, and because $FV(p) = \emptyset$ and $\mathtt{dom}(\eta) = \mathit{FMV}(\ell) \cup \{Z_1, \ldots, Z_n\} = \mathit{FMV}(\ell) \cup \mathit{FMV}(p)$. That it is a formative chain follows by the initial selection of γ, as we assumed formative reductions to each $\ell_j\gamma$.

It is also a computable chain: clearly we have $t_i = p\eta$ in step i in the construction above. Suppose $p^\sharp \trianglerighteq_B v$ and η respects B, but $(\lambda y_1 \ldots y_n.v)\eta$ is not computable for $FV(v) = \{y_1, \ldots, y_n\}$ – so by Lemma 80, $v(\eta \cup \zeta)$ is not computable for some computable substitution ζ on domain $FV(v)$. Since the meta-variables Z_j do not occur applied in p, we can safely assume that B contains only conditions for the meta-variables in $\mathtt{dom}(\gamma)$. By renaming each Z_j back to x_j, we obtain that γ respects B and $t \trianglerighteq_B v'$ with $v = v'[x_1 := Z_1, \ldots, x_n := Z_n]$. But then $r\gamma \trianglerighteq_{A \cup B} t \trianglerighteq_{A \cup B} v'$ and γ respects $A \cup B$ and $v'(\gamma \cup \delta \cup \zeta)$ is non-computable. By minimality of the choice $t\ (A)$, we have $v' = t$, so $v = p^\sharp$. However, $p^\sharp\eta$ has a marked symbol $\mathtt{g}^\sharp$ as a head symbol, and thus cannot be reduced at the top; by C-computability of its immediate subterms, it is computable. □

We also prove the statement that $\mathcal{U}$-computability implies minimality:

Lemma 84. *Every $\mathcal{U}$-computable $(\mathcal{P}, \mathcal{R})$-dependency chain is minimal.*

Proof. Let $[(\rho_0, s_0, t_0), (\rho_1, s_1, t_1), \ldots]$ be a $\mathcal{U}$-computable $(\mathcal{P}, \mathcal{R})$-chain and let $i \in \mathbb{N}$; we must prove that the strict subterms of t_i are terminating under $\Rightarrow_{\mathcal{R}}$. By definition, since $\trianglerighteq_\emptyset$ is a reflexive relation, t_i is $C_{\mathcal{U}}$-computable where $C_{\mathcal{U}}$ is given by Thm. 13 for a relation $\Rightarrow_{\mathcal{U}} \supseteq \Rightarrow_{\mathcal{R}}$. By Lemma 71(2), t_i is therefore terminating under $\Rightarrow_{\mathcal{U}}$, so certainly under $\Rightarrow_{\mathcal{R}}$ as well. The strict subterms of a terminating term are all terminating. □

B.2 Static dependency pairs: the inverse result

In this section, we prove Thm. 34, which states that the existence of certain kinds of dependency chains implies non-termination of the original AFSM. This is not a true inverse of Thm. 37 or even Thm. 32: as observed in the text, there are terminating AFSMs that do admit an infinite chain. It does, however, give us a way to use the DP framework to prove non-termination in some cases.

We begin by exploring the connection between $\trianglerighteq_\beta$ and reduction steps. Note that this result is *not* limited to PA-AFP AFSMs, so in the following we do not assume that the rules or dependency pairs involved satisfy arity restrictions.

Lemma 85. *Let s, t be meta-terms and suppose $s \trianglerighteq_A t$ for some set A of meta-variable conditions. Then for any substitution γ that respects A and has a finite domain with $FMV(s) \subseteq \mathtt{dom}(\gamma) \subseteq \mathcal{M}$: $s\gamma\ (\blacktriangleright \cup \Rightarrow_\beta)^*\ t\gamma$.*

Proof. By induction on the definition of $\trianglerighteq_A$. Consider the last step in its derivation.

- If $s = t$ then $s\gamma = t\gamma$.
- If $s = \lambda x.u$ and $u \trianglerighteq_A t$, then by α-conversion we can assume that $x \notin FV(\gamma(Z))$ for any $Z \in FMV(s)$. Thus, $s\gamma = \lambda x.(u\gamma) \blacktriangleright u\gamma\ (\blacktriangleright \cup \Rightarrow_\beta)^*\ t\gamma$ by the induction hypothesis.
- If $s = (\lambda x.u)\ s_0 \cdots s_n$ and $u[x := s_0]\ s_1 \cdots s_n \trianglerighteq_A t$, then by α-conversion we can safely assume that x is fresh w.r.t. γ as above; thus, $s\gamma = (\lambda x.(u\gamma))\ (s_0\gamma) \cdots (s_n\gamma) \Rightarrow_\beta (u\gamma[x := s_0\gamma])\ (s_1\gamma) \cdots (s_n\gamma) = (u[x := s_0]\ s_1 \cdots s_n)\gamma$, which reduces to $t\gamma$ by the induction hypothesis.
- If $s = u\ s_1 \cdots s_n$ for u an abstraction, variable, function symbol or meta-variable application, and $s_i \trianglerighteq_A t$, then $s\gamma = (u\gamma)\ (s_1\gamma) \cdots (s_n\gamma) \blacktriangleright s_i\gamma\ (\blacktriangleright \cup \Rightarrow_\beta)^*\ t\gamma$ by the induction hypothesis.
- If $s = Z\langle t_1, \dots, t_k\rangle\ s_1 \cdots s_n$ and $t_i \trianglerighteq_A t$ for some $1 \le i \le k$ with $(Z : i) \in A$, then we can write $\gamma(Z) = \lambda x_1 \dots x_n.w$ (where $n \le k$), and $s\gamma = w[x_1 := t_1\gamma, \dots, x_n := t_n\gamma]\ (t_{n+1}\gamma) \cdots (t_k\gamma)$. Since γ respects A, either x_i occurs in w or $i > n$; therefore $\gamma(Z) \underline{\blacktriangleright}\ t_i\gamma$. We again complete by the induction hypothesis. □

In fact, the text is ambiguous regarding the definition of *SDP* when an AFSM is not properly applied, since *minar*(f) may not be uniquely defined. However, the result holds for *any* choice of *minar*(f). In the following lemma, we only use that the elements of *SDP*($\mathcal{R}$) are DPs $\mathtt{f}^\sharp\ \ell_1 \cdots \ell_k \Rrightarrow \mathtt{g}^\sharp\ p_1 \cdots p_i\ (A)$ where $\mathtt{f}^\sharp\ \ell_1 \cdots \ell_k \Rightarrow r$ is a rule and there exist $p_{i+1} \dots p_n$ such that $r \trianglerighteq_A \mathtt{g}\ p_1 \cdots p_n$.

Lemma 86. *For $\ell^\sharp \Rrightarrow p^\sharp\ (A) \in SDP(\mathcal{R})$ such that $FMV(p) \subseteq FMV(\ell)$, and substitution γ on domain $FMV(\ell)$ such that γ respects the meta-variable conditions in A: both $\ell\gamma$ and $p\gamma$ are terms and $\ell\gamma\ (\Rightarrow_\mathcal{R} \cup \rhd)^+\ p\gamma$.*

Proof. By definition of *SDP* and the fact that no fresh meta-variables occur on the right, there is a rule $\ell \Rightarrow r$ such that $p\ (A) \in \mathsf{cand}(r)$, so there are $r_1, \dots, r_n$ such that $r \trianglerighteq_A p\ r_1 \cdots r_n$. Clearly, we have $\ell\gamma \Rightarrow_\mathcal{R} r\gamma$ by that rule, and $r\gamma\ (\rhd \cup \Rightarrow_\mathcal{R})^*\ (p\ r_1 \cdots r_n)\gamma \trianglerighteq p\gamma$ by Lemma 85 (using that $\blacktriangleright$ is a sub-relation of $\rhd$). We are done because $\Rightarrow_\beta$ is included in $\Rightarrow_\mathcal{R}$. □

This allows us to draw the required conclusion:

Theorem 34. *For any AFSM $(\mathcal{F},\mathcal{R})$: if there is an infinite $(\mathit{SDP}(\mathcal{R}),\mathcal{R})$-chain $[(\rho_0,s_0,t_0),(\rho_1,s_1,t_1),\dots]$ with all ρ_i conservative, then $\Rightarrow_{\mathcal{R}}$ is non-terminating.*

Proof. Let $s_i^\flat, t_i^\flat$ denote the terms s_i, t_i with all $\sharp$ marks removed. An infinite $(\mathit{SDP}(\mathcal{R}),\mathcal{R})$-dependency chain that does not use any DPs where fresh meta-variables are introduced on the right-hand side provides a sequence (s_i,t_i) for $i \in \mathbb{N}$ such that for all i, $s_i^\flat \ (\Rightarrow_{\mathcal{R}} \cup \rhd)^+ \ t_i^\flat$ (by Lemma 86), and $t_i^\flat \Rightarrow_{\mathcal{R}}^* s_{i+1}^\flat$. Thus, we obtain an infinite $\Rightarrow_{\mathcal{R}} \cup \rhd$ sequence, which provides an infinite $\Rightarrow_{\mathcal{R}}$ sequence due to monotonicity of $\Rightarrow_{\mathcal{R}}$. □

B.3 Original static dependency pairs

Since the most recent work on static dependency pairs has been defined for a polymorphic variation of the HRS formalism, it is not evident from sight how our definitions relate. Here, we provide context by showing how the definitions from [34,46] apply to the restriction of HRSs that can be translated to AFSMs.

It should be noted that HRSs, as translated to AFSMs, should be seen as η-expanded rules; in practice, for $\ell \Rightarrow r$ we have that $\ell \trianglerighteq s$ or $r \trianglerighteq s$ implies that either s is an abstraction, or s has base type. This definition implies that the system is properly applied, but is much stronger. We will refer to this restriction as *fully applied*.

Definition 87. *An AFSM $(\mathcal{F},\mathcal{R})$ is plain function passing following [34] if:*

- *for all rules $\mathtt{f}\ \ell_1 \cdots \ell_m \Rightarrow r$ and all $Z \in \mathit{FMV}(r)$: if Z does not have base type, then there are variables $x_1,\dots,x_n$ and some i such that $\ell_i = \lambda x_1 \dots x_n.Z\langle x_{j_1},\dots,x_{j_k}\rangle$.*

An AFSM $(\mathcal{F},\mathcal{R})$ is plain function passing following [46] if:

- *for all rules $\mathtt{f}\ \ell_1 \cdots \ell_m \Rightarrow r$ and all $Z \in \mathit{FMV}(r)$: there are some variables $x_1,\dots,x_k$ and some $i \leq m$ such that $\ell_i \trianglerighteq_{\mathtt{safe}}^{[46]} Z\langle x_1,\dots,x_k\rangle$, where the relation $\trianglerighteq_{\mathtt{safe}}^{[46]}$ is given by:*
  - *$s \trianglerighteq_{\mathtt{safe}}^{[46]} s$,*
  - *$\lambda x.t \trianglerighteq_{\mathtt{safe}}^{[46]} s$ if $t \trianglerighteq_{\mathtt{safe}}^{[46]} s$,*
  - *$x\ t_1 \cdots t_n \trianglerighteq_{\mathtt{safe}}^{[46]} s$ if $t_i \trianglerighteq_{\mathtt{safe}}^{[46]} s$ for some i with $x \in \mathcal{V} \setminus FV(t_i)$*
  - *$\mathtt{f}\ t_1 \cdots t_n \trianglerighteq_{\mathtt{safe}}^{[46]} s$ if $t_i \trianglerighteq_{\mathtt{safe}}^{[46]} s$ for some t_i of base type.*[7]

In addition, in both cases right-hand sides of rules are assumed to be presented in β-normal form and are fully applied.

[7] The authors of [46] refer to such subterms as *accessible*. We do not use this terminology, as it does not correspond to the accessibility notion in [8,9] which we follow here. In particular, the accessibility notion we use considers the relation $\succeq_+^{\mathcal{S}}$, which corresponds to the positive/negative inductive types in [8,9]. This is not used in [46].

The definitions of PFP in [34,46] also capture some non-pattern HRSs, but these cannot be represented as AFSMs. Note that the key difference between $\trianglerighteq^{[46]}_{\texttt{safe}}$ and $\trianglerighteq$ for patterns is that the former is not allowed to descend into a non-base argument of a function symbol. The same difference applies when comparing $\trianglerighteq^{[46]}_{\texttt{safe}}$ with $\trianglerighteq_{\texttt{acc}}$: $\trianglerighteq^{[46]}_{\texttt{safe}}$ also cannot descend into the accessible higher-order arguments.

Example 88. The rules from Ex. 6 are PFP following both definitions. The rules from Ex. 17 are PFP following [46] but not following [34]. The rules from Ex. 8 are not PFP in either definition, since $\texttt{lim}\ F \trianglerighteq^{[46]}_{\texttt{safe}} F$ does not hold (although they *are* AFP).

For a PFP AFSM, static dependency pairs are then defined as *pairs* $\ell^\sharp \Rrightarrow \texttt{f}^\sharp\ p_1 \cdots p_m$. This allows for a very simple notion of chains, even closer to the one in the first-order setting than our Def. 30.

Definition 89. *A static dependency chain following [34,46] is an infinite sequence* $[(\ell_0 \Rrightarrow p_0, \gamma_0), (\ell_1 \Rrightarrow p_1, \gamma_1), \dots]$ *where* $p_i\gamma_i \Rightarrow^*_{\mathcal{R}} \ell_{i+1}\gamma_{i+1}$ *for all* i*. It is* minimal *if each* $p_i\gamma_i$ *is terminating under* $\Rightarrow_{\mathcal{R}}$*.*

Both papers present a counterpart of Theorems 32 and 37 that roughly translates to the following:

Theorem 90 ([34,46]). *Let* $\mathcal{R}$ *be plain function passing following either definition in Def. 87. Let* $\mathcal{P} = \{\ell^\sharp \Rrightarrow \texttt{f}^\sharp\ p_1 \cdots p_m \mid \ell \Rightarrow r \in \mathcal{R} \wedge r \trianglerighteq \texttt{f}\ p_1 \cdots p_m \wedge \texttt{f} \in \mathcal{D} \wedge m = arity(\texttt{f})\}$*. If* $\Rightarrow_{\mathcal{R}}$ *is non-terminating, then there is an infinite minimal static dependency chain with all* $\ell_i \Rrightarrow p_i \in \mathcal{P}$*.*

Note that the chains are proved *minimal*, but not *computable* (which is a new definition in the current paper).

However, there is no counterpart to Thm. 34: this result relies on the presence of meta-variable conditions, which are not present in the static DPs from the literature.

Note that $\trianglerighteq_{\texttt{acc}}$ corresponds to $\trianglerighteq^{[46]}_{\texttt{safe}}$ (from Def. 87) if $\succeq^{\mathcal{S}}$ equates all sorts (as then always $Acc(\texttt{f}) = \{$ the indices of all base type arguments of $\texttt{f}\}$). Thus, Def. 18 includes both notions from Def. 87.

C Dependency pair processors

In this appendix, we prove the soundness – and where applicable completeness – of all DP processors defined in the text.

We first observe:

Lemma 91. *If Proc maps every DP problem to a set of problems such that for all* $(\mathcal{P}', \mathcal{R}', m', f') \in Proc(\mathcal{P}, \mathcal{R}, m, f)$ *we have that* $\mathcal{P}' \subseteq \mathcal{P}$*,* $\mathcal{R}' \subseteq \mathcal{R}$*,* $m' \succeq m$ *and* $f' = f$*, then Proc is complete.*

Proof. $\mathit{Proc}(\mathcal{P},\mathcal{R},m,f)$ is never `NO`. Suppose $\mathit{Proc}(\mathcal{P},\mathcal{R},m,f)$ contains an infinite element $(\mathcal{P}',\mathcal{R}',m',f')$; we must prove that then $(\mathcal{P},\mathcal{R},m,f)$ is infinite as well. This is certainly the case if $\Rightarrow_{\mathcal{R}}$ is non-terminating, so assume that $\Rightarrow_{\mathcal{R}}$ is terminating. Then certainly $\Rightarrow_{\mathcal{R}'} \subseteq \Rightarrow_{\mathcal{R}}$ is terminating as well, so $(\mathcal{P}',\mathcal{R}',m',f')$ can be infinite only because there exists an infinite $(\mathcal{P}',\mathcal{R}')$-chain that is $\mathcal{U}$-computable if $m' = \mathtt{computable}_{\mathcal{U}}$, minimal if $m' = \mathtt{minimal}$ and formative if $f' = \mathtt{formative}$. By definition, this is also a $(\mathcal{P},\mathcal{R})$-dependency chain, which is formative if $f = f' = \mathtt{formative}$. Since $\Rightarrow_{\mathcal{R}}$ is terminating, this chain is also minimal. If we have $m = \mathtt{computable}_{\mathcal{U}}$, then also $m' = \mathtt{computable}_{\mathcal{U}}$ (since $\mathtt{computable}_{\mathcal{U}}$ is maximal under $\succeq$) and the chain is indeed $\mathcal{U}$-computable. □

C.1 The dependency graph

The dependency graph processor lets us split a DP problem into multiple smaller ones. To prove soundness of its main processor, we first prove a helper result.

Lemma 92. *Let $M = (\mathcal{P},\mathcal{R},m,f)$ and G_θ an approximation of its dependency graph. Then for every infinite M-chain $[(\rho_0,s_0,t_0),(\rho_1,s_1,t_1),\dots]$ there exist $n \in \mathbb{N}$ and a cycle C in G_θ such that for all $i > n$: $\theta(\rho_i) \in C$.*

Proof. We claim (**): for all $i \in \mathbb{N}$, there is an edge from $\theta(\rho_i)$ to $\theta(\rho_{i+1})$. By definition of *approximation*, the claim follows if DG has an edge from ρ_i to ρ_{i+1}. But this is obvious: by definition of a chain, if $[(\rho_0,s_0,t_0),(\rho_1,s_1,t_1),\dots]$ is a dependency chain, then so is $[(\rho_i,s_i,t_i),(\rho_{i+1},s_{i+1},t_{i+1})]$.

Now, having (**), the chain traces an infinite path in G_θ. Let C be the set of nodes that occur infinitely often on this path; then for every node d that is not in C, there is an index n_d after which $\theta(\rho_i)$ is never d anymore. Since G_θ is a finite graph, we can take $n := \max(\{n_d \mid d \text{ a node in } G_\theta \wedge d \notin C\})$. Now for every pair $d,b \in C$: because they occur infinitely often, there is some $i > n$ with $\theta(\rho_i) = d$ and there is $j > i$ with $\theta(\rho_j) = b$. Thus, by (**) there is a path in G_θ from d to b. Similarly, there is a path from b to d. Hence, they are on a cycle. □

Note that to find a chain with all $\theta(\rho_i) \in C$, we do not need to modify the original chain at all: the satisfying chain is a tail of the original chain. Hence, the same flags apply to the resulting chain. This makes it very easy to prove correctness of the main processor:

Theorem 45 (Dependency graph processor). *The processor Proc_{G_θ} that maps a DP problem $M = (\mathcal{P},\mathcal{R},m,f)$ to $\{(\{\rho \in \mathcal{P} \mid \theta(\rho) \in C_i\},\mathcal{R},m,f) \mid 1 \leq i \leq n\}$ if G_θ is an approximation of the dependency graph of M and $C_1,\dots,C_n$ are the (nodes of the) non-trivial strongly connected components (SCCs) of G_θ, is both sound and complete.*

Proof. Completeness follows by Lemma 91. Soundness follows because if $(\mathcal{P},\mathcal{R},m,f)$ admits an infinite chain, then by Lemma 92 there is a cycle C such that a tail of this chain is mapped into C. Let C' be the strongly connected component in which C lies, and $\mathcal{P}' = \{\rho \in \mathcal{P} \mid \theta(\rho) \in C'\}$. Then clearly the same tail lies in $\mathcal{P}'$, giving an infinite $(\mathcal{P}',\mathcal{R},m,f)$-chain, and $(\mathcal{P}',\mathcal{R},m,f)$ is one of the elements of the set returned by the dependency graph processor. □

The dependency graph processor is essential to prove termination in our framework because it is the only processor defined so far that can map a DP problem to $\emptyset$.

C.2 Processors based on reduction triples

Theorem 49 (Basic reduction triple processor). *Let $M = (\mathcal{P}_1 \uplus \mathcal{P}_2, \mathcal{R}, m, f)$ be a DP problem. If $(\succsim, \succcurlyeq, \succ)$ is a reduction triple such that*

1. *for all $\ell \Rightarrow r \in \mathcal{R}$, we have $\ell \succsim r$;*
2. *for all $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1$, we have $\ell \succ p$;*
3. *for all $\ell \Rrightarrow p\ (A) \in \mathcal{P}_2$, we have $\ell \succcurlyeq p$;*

then the processor that maps M to $\{(\mathcal{P}_2, \mathcal{R}, m, f)\}$ is both sound and complete.

Proof. Completeness follows by Lemma 91. Soundness follows because every infinite $(\mathcal{P}_1 \uplus \mathcal{P}_2, \mathcal{R})$-chain $[(\rho_0, s_0, t_0), (\rho_1, s_1, t_1), \ldots]$ with $\mathcal{P}_1, \mathcal{P}_2, \mathcal{R}$ satisfying the given properties induces an infinite $\succ \cup \succcurlyeq \cup \succsim$ sequence, and every occurrence of a DP in $\mathcal{P}_1$ in the chain corresponds to a $\succ$ step in the sequence. By compatibility of the relations, well-foundedness guarantees that there can only be finitely many such steps, so there exists some n such that $[(\rho_n, s_n, t_n), (\rho_{n+1}, s_{n+1}, t_{n+1}), \ldots]$ is an infinite $(\mathcal{P}_2, \mathcal{R})$-chain.

To see that we indeed obtain the sequence, let $i \in \mathbb{N}$. Denote $\rho_i := \ell \Rrightarrow\ p\ (A)$, and let γ be a substitution on domain $FMV(\ell) \cup FMV(p)$ such that $s_i = \ell\gamma$ and $t_i = p\gamma$. Meta-stability gives us that $s_i = \ell\gamma\ (\succcurlyeq \cup \succ)\ p\gamma = t_i$. As $\Rightarrow_{\mathcal{R}}$ is included in $\succsim$ by meta-stability and monotonicity, and because $t_i \Rightarrow_{\mathcal{R}}^* s_{i+1}$, we have $t_i \succsim s_{i+1}$. Thus, $s_i (\succcurlyeq \cup \succ) \cdot \succsim s_{i+1}$. Moreover, a $\succ$ step is used if $\rho_i \in \mathcal{P}_1$. □

Now that we have seen a basic processor using reduction triples, soundness of the base-type processor presented in the text follows easily.

Theorem 52 (Reduction triple processor). *Let* Bot *be a set $\{\bot_\sigma : \sigma \mid \sigma$ a type$\} \subseteq \mathcal{F}^\sharp$ of unused constructors, $M = (\mathcal{P}_1 \uplus \mathcal{P}_2, \mathcal{R}, m, f)$ a DP problem and $(\succsim, \succcurlyeq, \succ)$ a reduction triple such that: (a) for all $\ell \Rightarrow r \in \mathcal{R}$, we have $\ell \succsim r$; and (b) for all $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1 \uplus \mathcal{P}_2$ with $\ell : \sigma_1 \to \ldots \to \sigma_m \to \iota$ and $p : \tau_1 \to \ldots \to \tau_n \to \kappa$ we have, for fresh meta-variables $Z_1 : \sigma_1, \ldots, Z_m : \sigma_m$:*

- *$\ell\ Z_1 \cdots Z_m \succ p\ \bot_{\tau_1} \cdots \bot_{\tau_n}$ if $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1$*
- *$\ell\ Z_1 \cdots Z_m \succcurlyeq p\ \bot_{\tau_1} \cdots \bot_{\tau_n}$ if $\ell \Rrightarrow p\ (A) \in \mathcal{P}_2$*

Then the processor that maps M to $\{(\mathcal{P}_2, \mathcal{R}, m, f)\}$ is both sound and complete.

Proof. Completeness follows by Lemma 91. Soundness follows by soundness of Thm. 49: let $(\succsim, \succcurlyeq, \succ)$ be a reduction triple satisfying the requirements above, and for $R \in \{\succcurlyeq, \succ\}$ define R' as follows: for $s : \sigma_1 \to \ldots \to \sigma_m \to \iota$ and $t : \tau_1 \to \ldots \to \tau_n \to \kappa$, let $s\ R'\ t$ if for all $u_1 : \sigma_1, \ldots, u_m : \sigma_m$ there exist $w_1 : \tau_1, \ldots, w_n : \tau_n$ such that $s\ u_1 \cdots u_m\ R\ t\ w_1 \cdots w_n$. We claim that $(\succsim, \succcurlyeq', \succ')$ is a reduction triple satisfying the requirements of Thm. 49, which implies soundness of the present processor.

It is clear that $\succcurlyeq'$ and $\succ'$ satisfy the requirements of Thm. 49: if $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1$, then for any $u_1, \dots, u_m$ we let $w_1 := \bot_{\tau_1}, \dots, w_n := \bot_{\tau_n}$ and have $\ell\ u_1 \cdots u_m \succ p\ w_1 \cdots w_n$ by meta-stability of $\succ$; the same holds for $\succcurlyeq$. It remains to be seen that $\succ'$ and $\succcurlyeq'$ are both transitive and meta-stable, that $\succcurlyeq'$ is reflexive and that $\succ'$ is well-founded.

- Meta-stability: given that $\ell \succ' p$ and γ is a substitution on domain $FMV(\ell) \cup FMV(p)$, we must see that $\ell\gamma \succ' p\gamma$ (the case for $\succcurlyeq'$ follows in the same way). Let $u_1, \dots, u_m$ be arbitrary terms and $\delta := \gamma \cup [Z_1 := u_1, \dots, Z_m := u_m]$; then $(\ell\ Z_1 \cdots Z_m)\delta \succ (p\ \bot_{\tau_1} \cdots \bot_{\tau_n})\delta$, so indeed $\ell\delta = \ell\gamma \succ' p\gamma$.
- Transitivity: if $s \succ t \succ v$ then for all $\boldsymbol{u}$ there exist $\boldsymbol{w}$ such that $s\ \boldsymbol{u} \succ t\ \boldsymbol{w}$, and for all $\boldsymbol{w}$ there exist $\boldsymbol{q}$ such that $t\ \boldsymbol{w} \succ v\ \boldsymbol{q}$; thus, also $s\ \boldsymbol{u} \succ v\ \boldsymbol{q}$. The case for $\succcurlyeq$ is similar.
- Reflexivity of $\succcurlyeq'$: always $s \succcurlyeq' s$ since for all $u_1, \dots, u_m$ we have $s\ \boldsymbol{u} \succcurlyeq s\ \boldsymbol{u}$.
- Well-foundedness of $\succ'$: suppose $s_1 \succ' s_2 \succ' \dots$ and let $\boldsymbol{u_1}$ be a sequence of variables; we find $\boldsymbol{u_2}, \boldsymbol{u_3}, \dots$ such that $s_1\ \boldsymbol{u_1} \succ' s_2\ \boldsymbol{u_2} \succ' \dots$ as in the case for transitivity. □

C.3 Rule removal without search for orderings

There is very little to prove: the importance is in the definition.

Theorem 58 (Formative rules processor). *For a formative rules approximation FR, the processor $Proc_{FR}$ that maps a DP problem $(\mathcal{P}, \mathcal{R}, m, \texttt{formative})$ to $\{(\mathcal{P}, FR(\mathcal{P}, \mathcal{R}), m, \texttt{formative})\}$ is both sound and complete.*

Proof. Completeness follows by Lemma 91. Soundness follows by definition of a formative rules approximation (a formative infinite $(\mathcal{P}, \mathcal{R})$-dependency chain can be built using only rules in $FR(\mathcal{P}, \mathcal{R})$). □

The practical challenge lies in proving that a given formative rules approximation really is one. The definition of a good approximation function is left to future work.

C.4 Subterm criterion processors

Next, we move on to the subterm processors. We first present the basic one – which differs little from its first-order counterpart, but is provided for context.

Theorem 61 (Subterm criterion processor). *The processor $Proc_{\texttt{subcrit}}$ that maps a DP problem $(\mathcal{P}_1 \uplus \mathcal{P}_2, \mathcal{R}, m, f)$ with $m \succeq \texttt{minimal}$ to $\{(\mathcal{P}_2, \mathcal{R}, m, f)\}$ if a projection function ν exists such that $\overline{\nu}(\ell) \rhd \overline{\nu}(p)$ for all $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1$ and $\overline{\nu}(\ell) = \overline{\nu}(p)$ for all $\ell \Rrightarrow p\ (A) \in \mathcal{P}_2$, is sound and complete.*

Proof. Completeness follows by Lemma 91. Soundness follows because an infinite $(\mathcal{P}, \mathcal{R}, m, f)$-chain with the properties above induces an infinite sequence $\overline{\nu}(s_0) \unrhd \overline{\nu}(t_0) \Rightarrow_{\mathcal{R}}^* \overline{\nu}(s_1) \unrhd \overline{\nu}(t_1) \Rightarrow_{\mathcal{R}}^* \dots$. Since the chain is minimal (either because

$m = \texttt{minimal}$, or by Lemma 84 if $m = \texttt{computable}_{\mathcal{U}}$), $\overline{\nu}(p_0)$ is terminating, and therefore it is terminating under $\Rightarrow_{\mathcal{R}} \cup \rhd$. Thus, there is some index n such that for all $i \geq n$: $\overline{\nu}(s_i) = \overline{\nu}(t_i) = \overline{\nu}(s_{i+1})$. But this can only be the case if $\overline{\nu}(\ell_i) = \overline{\nu}(p_i)$. But then the tail of the chain starting at position n does not use any pair in $\mathcal{P}_1$, and is therefore an infinite $(\mathcal{P}_2, \mathcal{R}, m, f)$-chain. □

We now turn to the proof of the computable subterm criterion processor. This proof is very similar to the one for the normal subterm criterion, but it fundamentally uses the definition of a computable chain.

Theorem 63 (Computable subterm criterion processor). *The processor $Proc_{\texttt{statcrit}}$ that maps a DP problem $(P_1 \uplus \mathcal{P}_2, \mathcal{R}, \texttt{computable}_{\mathcal{U}}, f)$ to $\{(\mathcal{P}_2, \mathcal{R}, \texttt{computable}_{\mathcal{U}}, f)\}$ if a projection function ν exists such that $\overline{\nu}(\ell) \sqsupset \overline{\nu}(p)$ for all $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1$ and $\overline{\nu}(\ell) = \overline{\nu}(p)$ for all $\ell \Rrightarrow p\ (A) \in \mathcal{P}_2$, is sound and complete. Here, $\sqsupset$ is the relation on base-type terms with $s \sqsupset t$ if $s \neq t$ and (a) $s \trianglerighteq_{\texttt{acc}} t$ or (b) a meta-variable Z exists with $s \trianglerighteq_{\texttt{acc}} Z\langle x_1, \ldots, x_k\rangle$ and $t = Z\langle t_1, \ldots, t_k\rangle\ s_1 \cdots s_n$.*

Proof. Completeness follows by Lemma 91. Soundness follows because, for $C := C_{\mathcal{U}}$ the computability predicate corresponding to $\Rightarrow_S$, an infinite $(\mathcal{P}, \mathcal{R}, \texttt{computable}_{\mathcal{U}}, f)$-chain induces an infinite $\Rrightarrow_C \cup \Rightarrow_{\mathcal{R}}$ sequence starting in the C-computable term $\overline{\nu}(s_1)$, with always $s_i (\Rrightarrow_C \cup \Rightarrow_{\mathcal{R}})^* t_i$ if $\rho_i \in \mathcal{P}_1$ and $\overline{\nu}(s_i) = \overline{\nu}(t_i)$ if $\rho_i \in \mathcal{P}_2$; like in the proof of the subterm criterion, this proves that the chain has a tail that is a $(\mathcal{P}_2, \mathcal{R}, \texttt{computable}_{\mathcal{U}}, f)$-chain because, by definition of C, $\Rightarrow_{\mathcal{R}} \cup \Rrightarrow_C$ is terminating on C-computable terms.

It remains to be seen that we indeed have $\overline{\nu}(s_i)\ (\Rrightarrow_C \cup \Rightarrow_{\mathcal{R}})^+ \overline{\nu}(t_i)$ whenever $\rho_i \in \mathcal{P}_1$. So suppose that ρ is a dependency pair $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1$ such that $\overline{\nu}(\ell) \sqsupset \overline{\nu}(p)$; we must see that $\overline{\nu}(\ell\gamma)\ (\Rrightarrow_C \cup \Rightarrow_\beta)^+ \overline{\nu}(p\gamma)$ for any substitution γ on domain $FMV(\ell) \cup FMV(r)$ such that $v\gamma$ is C-computable for all v, B such that $r \trianglerighteq_B v$ and γ respects B.

Write $\ell = \texttt{f}\ \ell_1 \cdots \ell_k$ and $p = \texttt{g}\ p_1 \cdots p_n$; then $\overline{\nu}(\ell\gamma) = \ell_{\nu(\texttt{f})}\gamma$ and $\overline{\nu}(p\gamma) = p_{\nu(\texttt{g})}\gamma$. Since, by definition of a dependency pair, ℓ is closed, we also have $FV(\ell_{\nu(\texttt{f})}) = \emptyset$. Consider the two possible reasons why $\ell_{\nu(\texttt{f})} \sqsupset p_{\nu(\texttt{g})}$.

- $\ell_{\nu(\texttt{f})} \trianglerighteq_{\texttt{acc}} p_{\nu(\texttt{g})}$: since both sides have base type by assumption and $\ell_{\nu(\texttt{f})}$ is closed, by Lemma 77 also $\ell_{\nu(\texttt{f})}\gamma\ (\Rrightarrow_C \cup \Rightarrow_\beta)^* p_{\nu(\texttt{g})}\gamma$.
- $\ell_{\nu(\texttt{f})} \trianglerighteq_{\texttt{acc}} Z\langle x_1, \ldots, x_k\rangle$ and $p_{\nu(\texttt{g})} = Z\langle u_1, \ldots, u_k\rangle\ v_1 \cdots v_n$: denote $\gamma(Z) = \lambda x_1 \ldots x_k.q$ and also $\gamma(Z) \approx_k \lambda x_1 \ldots x_k.q'$. Then we can write $q = \lambda x_{k+1} \ldots x_i.q''$ as well as $q' = q''\ x_{i+1} \cdots x_k$ for some $k \leq i \leq k$. Moreover:

$$p_{\nu(\texttt{g})}\gamma = q'[x_1 := u_1\gamma, \ldots, x_k := u_k\gamma]\ v_1\gamma \cdots v_n\gamma$$

  By definition of an $\mathcal{U}$-computable chain, $v_j\gamma$ is computable for each $1 \leq j \leq n$, and $u_j\gamma$ is computable for each $1 \leq j \leq k$ such that $x_j \in FV(q')$. Write $v'_j := v_j\gamma$ and let $u'_j := u_j\gamma$ if $x_j \in FV(q')$, otherwise $u'_j :=$ a fresh variable; then all u'_j and v'_j are computable, and still:

$$\begin{aligned} & p_{\nu(\texttt{g})}\gamma \\ &= q'[x_1 := u'_1, \ldots, x_k := u'_k]\ v'_1 \cdots v'_n \\ &= q''[x_1 := u'_1, \ldots, x_i := u'_i]\ u'_{i+1} \cdots u'_{k'}\ v'_1 \cdots v'_n \end{aligned}$$

On the other hand, by Lemma 77 and the observation that $FV(\ell_{\nu(\mathtt{f})}) = \emptyset$, we have $\ell_{\nu(\mathtt{f})}\gamma\ (\Rrightarrow_C \cup \Rightarrow_\beta)^+\ q[x_1 := u'_1, \dots, x_k := u'_k]\ u'_{k+1} \cdots u'_k\ v'_1 \cdots v'_n$, and as $q = \lambda x_{k+1} \dots x_i.q''$ this term β-reduces to $q''[x_1 := u'_1, \dots, x_i := u'_i]\ u'_{i+1} \cdots u'_{k'}\ v'_1 \cdots v'_n = p_{\nu(\mathtt{g})}\gamma$. □

C.5 Non-termination

Soundness and completeness of the non-termination processor in Thm. 65 are both direct consequences of Def. 40 and Def. 41.

D Experimental results

Finally, while the main paper focuses on a theoretical exposition, we here present an experimental evaluation of the results in this paper. The work has been implemented in the second author's termination tool WANDA, using *higher-order polynomial interpretations* [15] and a *recursive path ordering* [29, Chapter 5] for reduction triples. Both methods rely on an encoding of underlying constraints into SAT. The search for a sort ordering, and a projection function for the subterm criterion, is also delegated to SAT. The non-termination processor has not been implemented (WANDA performs some loop analysis, but outside the DP framework; this is a planned future improvement), and the subterm criterion processor has been merged with the computable subterm criterion processor.

In addition to the results in this paper, WANDA includes a search for monotonic termination orderings outside the DP framework (using successive rule removal), a dynamic DP framework (following [31]), and a mechanism [14] within both DP frameworks to delegate some first-order parts of the AFSM to a first-order termination tool (here we use AProVE [19]).

We have evaluated the power of our techniques on the *Termination Problems Database* [49], version 10.5. Of the 198 benchmarks in the category *Higher Order Union Beta*, 153 are accessible function passing. Comparing the power of static DPs versus dynamic DPs or no DP framework gives the following results (where *Time* is the average runtime on success in seconds):

Technique	**Yes**	**Time**
Only rule removal	92	0.36
Static DPs with techniques from this paper	124	0.07
Dynamic DPs with techniques from this paper	132	0.53
Static DPs with delegation to a first-order prover	129	0.58
Dynamic DPs with delegation to a first-order prover	137	1.01
Static and dynamic DPs with delegation to a first-order prover	150	0.73
Non-terminating?	16	0.66

While static DPs have a slightly lower success rate than dynamic DPs, their evaluation is much faster, since they allow for greater modularity: the dynamic setting includes dependency pairs where the right-hand does not have a (marked) defined symbol at the head (e.g., $\mathtt{map}\ F\ (\mathtt{cons}\ H\ T) \Rrightarrow F\ H$), which make both the subterm criterion processor and the dependency graph processor harder to apply. The combination of static and dynamic DPs performs substantially better than either style alone: although the gains can be seen as modest if the size of the data set is not taken into account (153 versus 166 YES+NO, giving an 8% increase), the numbers indicate a 29% *decrease* in failure rate (from 45 to 32).

We have also compared the individual techniques in this paper by disabling them from the second test above. This gives the table below:

Disabled	**Yes**	**Time**
Formative rules	124	0.06
Usable rules	124	0.09
Subterm criterion	121	0.15
Graph	121	0.33
Reduction triples	92	0.01
Nothing	124	0.06

Note that none of the techniques individually give much power except for reduction triples: where one method is disabled, another can typically pick up the slack. If all processors except reduction triples are disabled, only 105 benchmarks are proved. Most processors do individually give a significant *speedup*. Only formative rules does not; this processor is useful in the dynamic setting, but does not appear to be so here.

Evaluation pages for these experiments are available at:

`https://www.cs.ru.nl/~cynthiakop/experiments/esop2019/`