WANDA – a Higher Order Termination Tool

Cynthia Kop
Radboud University, The Netherlands
https://www.cs.ru.nl/~cynthiakop/
c.kop@cs.ru.nl

Abstract

Wanda is a fully automatic termination analysis tool for higher-order term rewriting. In this paper, we will discuss the methodology used in Wanda. Most pertinently, this includes a higher-order dependency pair framework and a variation of the higher-order recursive path ordering, as well as some non-termination analysis techniques and delegation to a first-order tool. Additionally, we will discuss Wanda's internal rewriting formalism, and how to use Wanda in practice for systems in two different formalisms. We also present experimental results that consider both formalisms.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting

Keywords and phrases higher-order term rewriting, termination, automatic analysis, dependency pair framework, higher-order recursive path ordering

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.36

Category System Description

Supplementary Material A detailed experimental evaluation and the snapshot of Wanda used in this paper are available from: https://www.cs.ru.nl/~cynthiakop/experiments/fscd20.

Funding The author is supported by the NWO TOP project "ICHOR", NWO 612.001.803/7571.

Acknowledgements Thanks go to Carsten Fuhs both for proof-reading and for creating a customised version of AProVE which gives an explicit example term for non-termination; to Julian Nagele for using CSI^ho to translate the pattern HRSs in COPS to AFSMs; and to the anonymous reviewers of FSCD 2020 whose thorough feedback helped to improve the paper.

1 Introduction

Termination of term rewriting systems has been an area of active research for several decades. This concerns not only the analysis of pure term rewriting, but also many variants, such as context-sensitive [51], conditional [40] and higher-order [7] term rewriting. Since the introduction of the annual *International Termination Competition* [12], automated techniques in particular have flourished, with many strong provers competing against each other.

Compared to the core area of first-order term rewriting, *higher-order* term rewriting provides some unique challenges, for example due to bound variables. Nevertheless, several tools have participated in the higher-order category of the termination competition (Hot [4], THOR [8], Sol [28], SizeChangeTool [23], Wanda), each using different methods; these include both extensions of first-order techniques like recursive and semantic path orderings [30, 14, 29, 9] and dependency pairs [3, 38, 37], and also dedicated methods such as sized types [5].

Wanda, a tool built primarily around dependency pairs, has participated in this category since 2010 and won most years, including 2019. Wanda was also used as a termination back-end in the higher-order category of the 2019 *International Confluence Competition* [11], with both participants (ACPH [44] and CSI^ho [42]) delegating termination analysis to Wanda.

Despite this history, Wanda is not well-documented: no tool description has ever been formally published. Implementation choices *are* outlined in the author's PhD thesis [35] alongside termination techniques, but are not easily accessible as understanding these parts requires an understanding of the whole document. This has led to problems, as critical

details – such as the rewriting formalism Wanda employs, what Wanda actually does and how to use Wanda in different configurations or for different styles of rewriting – are hard to find. In addition, there have been substantial updates in recent years.

The present work will address this issue by presenting the usage of and the most important techniques used in Wanda. To start, the formalism of higher-order rewriting Wanda uses, AFSMs, is explained in §2, as well as its relation to other popular higher-order formalisms. Then we will discuss the non-termination and termination techniques used in Wanda (§3–5), The paper ends with experimental results, practical information and conclusions (§6–8).

Wanda is open-source, and is available at: `http://wandahot.sourceforge.net`. The snapshot that was used in the present paper, including all back-ends, is available from: `https://www.cs.ru.nl/~cynthiakop/experiments/fscd20/wanda2020.zip` .

Theoretical contribution. Although the focus is on Wanda, this paper also presents some theoretical results that were previously only published in the author's PhD thesis:

- a transformation from pattern HRSs [43] to Wanda's internal format, AFSMs;
- two simple non-termination techniques (§3.1–3.2);
- a new variation of the higher-order recursive path ordering suited to AFSMs (§4.2).

In addition, the results of §2.3 and §4.1, and the "dynamic" part of §5, were previously presented for a more restricted formalism and are here generalised to AFSMs. The remaining results in this paper connect and discuss existing work, and explain how it is used in Wanda.

2 Higher-order term rewriting using AFSMs

There is no single, unified approach to higher-order term rewriting; rather, there are several similar but not fully compatible systems. This is a problem, since users of various kinds of higher-order TRSs may be interested in termination, and it would be frustrating to adapt techniques and write different tools for each style. Therefore, Wanda uses a custom format, *AFSMs*, which several popular kinds of rewriting systems can be translated into. AFSMs (Algebraic Functional Systems with Meta-variables) are essentially simply-typed CRSs [32] and also largely correspond to the formalism in [6]. AFSMs are fully presented in [22].

2.1 Preliminaries: the AFSM formalism

Wanda operates on typed expressions, defined by Definitions 1 and 2.

▶ **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 $\sigma \to \tau$. Here, $\to$ is right-associative.*

Denoting ι, κ for a sort, all types have a unique form $\sigma_1 \to \ldots \to \sigma_m \to \iota$.

This definition does not have type variables, which occur in polymorphic styles of rewriting. Wanda *does* allow them as input, but since the implementation of most termination techniques does not support polymorphism, we will here consider only the simple types above.

▶ **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 (its arity). 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*	$s : \sigma \to \tau$ *and* $t : \sigma$
(F)	$\mathtt{f} : \sigma$	*if*	$\mathtt{f} : \sigma \in \mathcal{F}$	*(Λ)*	$\lambda x.s : \sigma \to \tau$	*if*	$x : \sigma \in \mathcal{V}$ *and* $s : \tau$

Meta-terms *are expressions whose type can be derived by the clauses above along with:*

(M) $Z[s_1, \ldots, s_k] : \sigma_{k+1} \to \ldots \to \sigma_m \to \iota$
if $Z : (\sigma_1 \to \ldots \to \sigma_k \to \ldots \to \sigma_m \to \iota,\ k) \in \mathcal{M}$ *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* $\sigma \in \mathcal{S}$*. Let* $\mathsf{head}(s) = \mathsf{head}(s_1)$ *if* $s = s_1\ s_2$*; otherwise* $\mathsf{head}(s) = s$*.*

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* $s_1 \trianglerighteq t$*.*

Note that 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 $Z[s_1, \ldots, s_k]$. *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. An example of a meta-term is $\lambda x.\lambda y.\mathtt{sin}\ Z[x]$. In the left-hand side of a rule, this meta-term stands for an arbitrary term of the form $\lambda x.\lambda y.\mathtt{sin}\ t$ where t may contain the bound variable x, but not the bound variable y. This is more fully defined in Definitions 4 and 5.

For rewriting, we will additionally employ *patterns*:

▶ **Definition 3** (Patterns). *A meta-term is a* pattern *if it has one of the forms* $Z[x_1, \ldots, x_k]$ *with all* x_i *distinct variables and* $Z : (\sigma, k) \in \mathcal{M}$ *for some* σ*;* $\lambda x.\ell$ *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, meta-variables are used for *matching* and variables are only used with *binders*. In terms, variables can occur both free and bound, and meta-variables cannot occur. Meta-variables originate in early forms of higher-order rewriting (e.g., [1, 32]), but have also been used in later formalisms (e.g., [6]). 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 [41], but without including a reversed β-reduction as part of matching.

In Wanda, function symbols are identified by their name, and variables and meta-variables by an integer index; using integers makes it very easy to allocate fresh variables when needed. The indexes are not shown to the user; instead a unique name is generated for printing.

▶ **Definition 4** (Substitution). *A* substitution γ *is a type-preserving mapping from a subset of* $\mathcal{V} \cup \mathcal{M}$ *(the* domain *of* γ*) to terms, typically denoted in a form* $\gamma = [b_1 := s_1, \ldots, b_n := s_n]$ *(here, the domain is* $\{b_1, \ldots, b_n\}$*). Substitutions may have infinite domain, but – denoting* $\mathtt{dom}(\gamma)$ *for the domain of* γ *– we require that 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 extended to a function from meta-terms to meta-terms as follows:

$$
\begin{array}{rcll}
x\gamma & = & \gamma(x) & \text{if} \quad x \in \mathcal{V} \cap \mathtt{dom}(\gamma) \\
x\gamma & = & x & \text{if} \quad x \in \mathcal{V} \setminus \mathtt{dom}(\gamma) \\
\mathtt{f}\gamma & = & \mathtt{f} & \text{if} \quad \mathtt{f} \in \mathcal{F} \\
(s\ t)\gamma & = & (s\gamma)\ (t\gamma) & \\
(\lambda x.s)\gamma & = & \lambda x.(s\gamma) & \text{if} \quad x \notin \mathtt{dom}(\gamma) \wedge x \notin \bigcup_{y \in \mathtt{dom}(\gamma)} FV(\gamma(y)) \\
Z[s_1, \ldots, s_k]\gamma & = & Z[s_1\gamma, \ldots, s_k\gamma] & \text{if} \quad Z \notin \mathtt{dom}(\gamma) \\
Z[s_1, \ldots, s_k]\gamma & = & t[x_1 := s_1\gamma, \ldots, x_k := s_k\gamma] & \text{if} \quad \gamma(Z) = \lambda x_1 \ldots x_k.t \\
Z[s_1, \ldots, s_k]\gamma & = & t[x_1 := s_1\gamma, \ldots, x_n := s_n\gamma]\ \ (s_{n+1}\gamma) \cdots (s_k\gamma) & \text{if} \quad \gamma(Z) = \lambda x_1 \ldots x_n.t \\
 & & & \quad \wedge\ n < k
\end{array}
$$

Note that substituting an abstraction is fully defined due to α-conversion and the requirement that there are infinitely many variables not occurring in the domain or range of γ. Moreover, for fixed k, any meta-term $\gamma(Z)$ can be written in the form $\lambda x_1 \dots x_n.t$ with either $n < k$ and t not an abstraction, or $n = k$ (and t unrestricted). Thus, this is well-defined.

Essentially, applying a substitution with meta-variables in its domain combines a substitution with a β-development. For example, $\mathtt{deriv}\ (\lambda x.\mathtt{sin}\ (F[x]))[F := \lambda y.\mathtt{plus}\ y\ x]$ equals $\mathtt{deriv}\ (\lambda z.\mathtt{sin}\ (\mathtt{plus}\ z\ x))$, and $X[\mathtt{0},\mathtt{nil}][X := \lambda x.\mathtt{map}\ (\lambda y.x)]$ equals $\mathtt{map}\ (\lambda y.\mathtt{0})\ \mathtt{nil}$. If $\mathtt{dom}(\gamma)$ contains all meta-variables in $FMV(s)$, then $s\gamma$ is a term.

▶ **Definition 5** (Rules and Rewriting). *A* rule *is a pair $\ell \Rightarrow r$ of closed meta-terms of the same type, where ℓ is a pattern of the form $\mathtt{f}\ \ell_1 \cdots \ell_n$ with $\mathtt{f} \in \mathcal{F}$, and $FMV(r) \subseteq FMV(\ell)$. For a set of rules $\mathcal{R}$, reduction is the smallest monotonic relation $\Rightarrow_{\mathcal{R}}$ on terms that includes:*

(Rule) $\quad \ell\gamma \quad \Rightarrow_{\mathcal{R}} \quad r\gamma \quad$ *for $\ell \Rightarrow r \in \mathcal{R}$, and γ a substitution with $\mathtt{dom}(\gamma) = FMV(\ell)$*

(Beta) $\quad (\lambda x.s)\ t \quad \Rightarrow_{\mathcal{R}} \quad s[x := t]$

Note that we can reduce at any position of a term, even below a λ. We write $s \Rightarrow_\beta t$ if $s \Rightarrow_{\mathcal{R}} t$ is derived using (Beta). A term s is terminating *under $\mathcal{R}$ 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 with $s \Rightarrow_{\mathcal{R}} t$, and* is β-normal *if there is no t with $s \Rightarrow_\beta t$. The relation $\Rightarrow_{\mathcal{R}}$ is terminating if all terms are terminating.*

Although the theory in [35] allows for $\mathcal{R}$ to be infinite (mostly with an eye on polymorphism), Wanda does not fully support this yet, so we will here limit interest to finite $\mathcal{R}$.

▶ **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 $\mathcal{R}_1$:

$$\begin{array}{rcl} \mathtt{map}\ (\lambda x.Z[x])\ \mathtt{nil} & \Rightarrow & \mathtt{nil} \\ \mathtt{map}\ (\lambda x.Z[x])\ (\mathtt{cons}\ H\ T) & \Rightarrow & \mathtt{cons}\ Z[H]\ (\mathtt{map}\ (\lambda x.Z[x])\ T) \end{array}$$

Then $\mathtt{map}\ (\lambda y.\mathtt{0})\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{nil}) \Rightarrow_{\mathcal{R}_1} \mathtt{cons}\ \mathtt{0}\ (\mathtt{map}\ (\lambda y.\mathtt{0})\ \mathtt{nil}) \Rightarrow_{\mathcal{R}_1} \mathtt{cons}\ \mathtt{0}\ \mathtt{nil}$. Note that the bound variable y does not need to occur in the body of $\lambda y.\mathtt{0}$ to be matched by $\lambda x.Z[x]$. However, note also that a term like $\mathtt{map}\ \mathtt{s}\ (\mathtt{cons}\ \mathtt{0}\ \mathtt{nil})$ *cannot* be reduced, because $\mathtt{s}$ does not match $\lambda x.Z[x]$. We could alternatively consider the rules $\mathcal{R}_2$:

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

In the previous example, we had $Z : (\mathtt{nat} \to \mathtt{nat}, 1) \in \mathcal{M}$; here, we have $Z : (\mathtt{nat} \to \mathtt{nat}, 0) \in \mathcal{M}$ (we will typically leave this implicit since the arity of meta-variables can be read off from the left-hand sides of the rules). Instead of meta-variable application $Z[x]$, we use explicit application $Z\ x$. Now we do have $\mathtt{map}\ \mathtt{s}\ (\mathtt{cons}\ \mathtt{0}\ \mathtt{nil}) \Rightarrow_{\mathcal{R}_2} \mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ (\mathtt{map}\ \mathtt{s}\ \mathtt{nil})$. However, now we will often need explicit β-reductions; e.g., $\mathtt{map}\ (\lambda y.\mathtt{0})\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{nil}) \Rightarrow_{\mathcal{R}_2} \mathtt{cons}\ ((\lambda y.\mathtt{0})\ (\mathtt{s}\ \mathtt{0}))\ (\mathtt{map}\ (\lambda y.\mathtt{0})\ \mathtt{nil}) \Rightarrow_\beta \mathtt{cons}\ \mathtt{0}\ (\mathtt{map}\ (\lambda y.\mathtt{0})\ \mathtt{nil})$.

Thus, AFSMs allow us to define essentially the same rules in multiple ways. This flexibility may seem redundant, but is necessary to enable the analysis of different styles of higher-order term rewriting, as we will see in §2.2. An AFSM is a pair $(\mathcal{T}(\mathcal{F}, \mathcal{V}), \Rightarrow_{\mathcal{R}})$ of a set of terms and a reduction relation on that set. To define an AFSM, it suffices to supply $\mathcal{F}$ and $\mathcal{R}$; types of (meta-)variables can be derived from context. This is what Wanda takes as input.

▶ **Example 7.** The first `map` rules from Example 6 can be given to Wanda, in a file `map.afsm`, which provides first the signature and then the rules:

```
nil : list
cons : nat -> list -> list
map : (nat -> nat) -> list -> list

map (/\x.Z[x]) nil => nil
map (/\x.Z[x]) (cons H T) => cons Z[H] (map (/\x.Z[x]) T)
```

Note: all identifiers (function symbols, variables and meta-variables) in `.afsm` files are expected to be alphanumeric. Characters such as +, - and _ are not allowed in names. The only exceptions are the exclamation mark symbol ('!') and the percentage symbol ('%'); the latter may only be used at the start of (meta-)variables.

2.2 Transformations

AFSMs are not meant to be interesting in their own right. Rather, they are defined to support termination proofs in multiple formalisms. Let us consider the two most relevant.

Higher-order Rewriting Systems (HRSs) [43] are one of the oldest styles of higher-order term rewriting. Here, rewriting is *modulo* $\Rightarrow_\beta$: for terms s, t in η-long β-normal form we have $s \Rightarrow_{\mathcal{R}} t$ if there exist a rule $\ell \Rightarrow r$ and a substitution γ such that $\ell\gamma \Rightarrow_\beta^* s$ and $r\gamma \Rightarrow_\beta^* t$. All terms are presented in η-long β-normal form, and rules are pairs of such terms (there are no meta-variables). The η-long form of a term s is obtained by repeatedly applying the step "$s \Rightarrow_\eta \lambda x.(s\ x)$" on all subterms of s where this can be done without creating a β-redex.

In general, the reduction relation $\Rightarrow_{\mathcal{R}}$ in an HRS is not computable, but practical examples typically consider *pattern HRSs* (PRSs), where for all rules $\ell \Rightarrow r$ and for all subterms $x\ \ell_1 \cdots \ell_m$ of the left-hand side with x a variable: each ℓ_i is the η-long form of a distinct bound variable. Pattern HRSs are translated to AFSMs in a natural way, by replacing free variables in the rules by meta-variables, and their applications by meta-applications.

▶ **Example 8.** Let us consider an example of a pattern HRS:

$$\begin{array}{rcl} \texttt{bind}\ (\texttt{return}\ x)\ (\lambda y.f\ y) & \Rightarrow & f\ x \\ \texttt{bind}\ x\ (\lambda y.\texttt{return}\ y) & \Rightarrow & x \\ \texttt{bind}\ (\texttt{bind}\ x\ (\lambda y.f\ y))\ (\lambda z.g\ z) & \Rightarrow & \texttt{bind}\ x\ (\lambda u.\texttt{bind}\ (f\ u)\ (\lambda v.g\ v)) \end{array}$$

It is translated to the following AFSM (meta-variables are indicated with capitals):

$$\begin{array}{rcl} \texttt{bind}\ (\texttt{return}\ X)\ (\lambda y.F[y]) & \Rightarrow & F[X] \\ \texttt{bind}\ X\ (\lambda y.\texttt{return}\ y) & \Rightarrow & X \\ \texttt{bind}\ (\texttt{bind}\ X\ (\lambda y.F[y]))\ (\lambda z.G[z]) & \Rightarrow & \texttt{bind}\ X\ (\lambda u.\texttt{bind}\ (F[u])\ (\lambda v.G[v])) \end{array}$$

This translated system has very similar behaviour to the original PRS, but there is a critical difference: the PRS is a relation on η-long β-normal terms, while the AFSM is generally considered as a relation on all terms. It turns out that the restriction to η-long terms does not affect termination, but the β-normalisation does ([35, Theorem 3.5]):

▶ **Lemma 9.** *The original PRS $(\mathcal{F}, \mathcal{R})$ is terminating if and only if the translated AFSM $(\mathcal{F}, \mathcal{R}')$ is terminating using a reduction strategy where $\Rightarrow_\beta$ is preferred to other steps.*

That is, we need to β-normalise terms after every reduction step. Wanda can test the property of termination with a $\Rightarrow_\beta$-first strategy by being invoked with a runtime argument `--betafirst` (e.g., `./wanda.exe --betafirst system.afsm`). As a side note, however, the examples where this requirement makes a difference are rare and typically artificial.

▶ **Example 10.** Let $\mathcal{F} = \{\mathtt{a} : \mathtt{o},\ \mathtt{f} : \mathtt{o} \to \mathtt{o},\ \mathtt{g} : ((\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}) \to \mathtt{o}\}$ and $\mathcal{R}$ given by:

$$\mathtt{f}\ \mathtt{a} \quad \Rightarrow \quad \mathtt{g}\ (\lambda x.\lambda y.x\ (\mathtt{f}\ y)) \qquad \mathtt{g}\ (\lambda x.\lambda y.h\ (\lambda z.x\ z)\ y) \quad \Rightarrow \quad h\ (\lambda z.\mathtt{a})\ (\mathtt{a})$$

This PRS is translated to an AFSM with the following rules $\mathcal{R}'$:

$$\mathtt{f}\ \mathtt{a} \quad \Rightarrow \quad \mathtt{g}\ (\lambda x.\lambda y.x\ (\mathtt{f}\ y)) \qquad \mathtt{g}\ (\lambda x.\lambda y.H[x,y]) \quad \Rightarrow \quad H[\lambda z.\mathtt{a},\ \mathtt{a}]$$

While the original PRS is terminating, the same does not hold for the translated AFSM: we have $\mathtt{f}\ \mathtt{a} \Rightarrow_{\mathcal{R}'} \mathtt{g}\ (\lambda x.\lambda y.x\ (\mathtt{f}\ y)) \Rightarrow_{\mathcal{R}'} (\lambda z.\mathtt{a})\ (\mathtt{f}\ \mathtt{a})$, where the last term has $\mathtt{f}\ \mathtt{a}$ as a subterm. In an AFSM, it is not mandatory to reduce the β-redex. Wanda concludes non-termination normally, but cannot find a proof or disproof if the `--betafirst` argument is provided.

Remark: Wanda has not been optimised for HRSs, and does not take advantage of the `--betafirst` argument other than avoiding false claims of non-termination. This is primarily due to a lack of motivating examples: the annual Termination Competition does not consider HRSs. However, since the *International Confluence Competition* [11] *does* consider HRSs, and comes with its own benchmark set, this situation is likely to change in the future.

Algebraic Functional Systems (AFSs) [29] are higher-order term rewriting systems with $\Rightarrow_\beta$ as a separate step (i.e., $\Rightarrow_\beta \subseteq \Rightarrow_{\mathcal{R}}$; unlike HRSs, β-steps are not implicitly done as part of other steps); this is the format used in the higher-order category of the International Termination Competition [12]. Rules in an AFS are pairs of *terms*, not *meta-terms*, and there is no pattern Another difference with AFSMs is that AFSMs use applicative (curried) notation while AFSs use a mixture of functional and applicative term formation; however, this difference is not significant, since – following [34, 35] – currying does not affect termination.

Using variables rather than meta-variables for matching is not important either: just replace all free variables by meta-variables. This gives rules like the "alternative" rules $\mathcal{R}_2$ in Example 6. However, the lack of a pattern restriction is very significant.

▶ **Example 11.** Let us consider an example of an AFS that cannot be naturally translated without violating pattern restrictions. We let $\mathcal{F} = \{\mathtt{new} : (\mathtt{N} \to \mathtt{A}) \to \mathtt{A}\}$ and $\mathcal{R}$ consist of:

$$\mathtt{new}\ (\lambda x.y) \quad \Rightarrow \quad y \qquad \mathtt{new}\ (\lambda x.\mathtt{new}\ (\lambda y.f\ x\ y)) \quad \Rightarrow \quad \mathtt{new}\ (\lambda x.\mathtt{new}\ (\lambda y.f\ y\ x))$$

Now, the left-hand sides *look* like patterns. Indeed, they satisfy the requirements for an HRS-pattern: the free variable f in the second rule is only applied to distinct bound variables. So if this was an HRS, we could translate it to the following AFSM:

$$\mathtt{new}\ (\lambda x.Y) \quad \Rightarrow \quad Y \qquad \mathtt{new}\ (\lambda x.\mathtt{new}\ (\lambda y.F[x,y])) \quad \Rightarrow \quad \mathtt{new}\ (\lambda x.\mathtt{new}\ (\lambda y.F[y,x]))$$

However, since the original system was an AFS, this is *not* equivalent. Unlike in HRSs, matching in AFSs is not modulo beta: like in AFSMs, s rewrites to t by rule $\ell \Rightarrow r$ if there exists a substitution γ such that $s = \ell\gamma$ and $t = r\gamma$. So, in the AFS, the subterm $f\ x\ y$ can *only* be instantiated by terms of the form $s\ x\ y$. An accurate translation of the second rule to AFSMs would simply replace $f\ x\ y$ by $F[]\ x\ y$, resulting in a non-pattern.

This is important because the AFSM above is non-terminating: $\mathtt{new}\ (\lambda x.\mathtt{new}\ (\lambda y.z))$ reduces to itself in one step because the meta-variable F can be instantiated by a substitution $\lambda x.\lambda y.z$. On the other hand, the original AFS is terminating, as we will see below.

A final difference is that, following [29], AFSs use polymorphic types. Wanda limits interest to simply-typed AFSs, which is what the Termination Competition uses. Polymorphic AFSs can be translated to polymorphic AFSMs, but this is not yet well-supported in Wanda.

Wanda accepts AFSs as input directly (using the xml format of the Termination Competition or a custom human-readable format). *Most* AFSs can be naturally translated into AFSMs just by replacing free variables by meta-variables; typically counterexamples look like they were meant as HRSs, but translated poorly into AFSs. For the examples that cannot be naturally translated, Wanda first applies the transformations in [34] to create patterns. This involves introducing fresh symbols $\mathtt{app}_i$ to replace some of the applications $s\ t$ by terms of the form $\mathtt{app}_i\ s\ t$. New rules may also be introduced, as for instance $\mathtt{f}\ (X\ Y)\ \mathtt{a} \Rightarrow \mathtt{f}\ (X\ \mathtt{b})\ Y$ is replaced by not only $\mathtt{f}\ (\mathtt{app}_i\ X\ Y)\ \mathtt{a} \Rightarrow \mathtt{f}\ (\mathtt{app}_i\ X\ \mathtt{b})\ Y$, but in addition potentially many rules of the form $\mathtt{f}\ (\mathtt{g}\ X_1 \cdots X_n\ Y)\ \mathtt{a} \Rightarrow \mathtt{f}\ (\mathtt{g}\ X_1 \cdots X_n\ \mathtt{a})\ Y$. This is exacerbated when the AFS is presented in curried (applicative) form rather than functional notation.

▶ **Example 12.** The AFS of Example 11 is translated to an AFSM with the following rules:

$$\begin{array}{rcl} \mathtt{new}\ (\lambda x.Y) & \Rightarrow & Y \\ \mathtt{new}\ (\lambda x.\mathtt{new}\ (\lambda y.\mathtt{app}\ F\ x\ y)) & \Rightarrow & \mathtt{new}\ (\lambda x.\mathtt{new}\ (\lambda y.\mathtt{app}\ F\ u\ z)) \\ \mathtt{app}\ F\ X & \Rightarrow & F\ X \end{array}$$

This AFSM can be proved terminating by Wanda's recursive path ordering (§4.2) in combination with dependency pairs (§5).

It is worth noting that the transformations needed to translate an AFS to an AFSM with equivalent behaviour can sometimes cause the system to become much more difficult to analyse, both due to the inclusion of explicit "application" symbols in the rules and the addition of potentially many new rules. For this reason, Wanda uses the following approach:

- create both an accurate translation and an *overestimation* of the AFS (so that termination of the overestimation implies termination of the original system, but not the reverse); this results in translations like those given in Example 11;
- try to prove non-termination using the accurate translation;
- try to prove termination using the overestimation;
- if this fails, try to prove termination using the accurate translation.

2.3 Uncurrying

Following [35, §2.3.1] and [34, §7], *uncurrying* does not affect termination provided the rules are (essentially) unchanged. That is, we can denote both rules and terms in a functional notation, but only if the number of arguments is respected in each rule. To be exact:

▶ **Lemma 13.** *Let $(\mathcal{F}, \mathcal{R})$ be an AFSM, and let $minar(\mathtt{f})$ denote the largest number k such that (1) the type of $\mathtt{f}$ allows $\mathtt{f}$ to be applied to at least k arguments, and (2) every occurrence of $\mathtt{f}$ in $\mathcal{R}$ is applied to at least k arguments. Then $\Rightarrow_{\mathcal{R}}$ is non-terminating if and only if there is an infinite reduction $s_1 \Rightarrow_{\mathcal{R}} s_2 \Rightarrow_{\mathcal{R}} \ldots$ where, in every term s_i, each symbol $\mathtt{f}$ always occurs with at least $minar(\mathtt{f})$ arguments.*

For example, in Example 6, $minar(\mathtt{s}) = 1$ and $minar(\mathtt{cons}) = minar(\mathtt{map}) = 2$; thus, we do not need to consider terms such as $\mathtt{map}\ \mathtt{s}\ (\mathtt{cons}\ \mathtt{0}\ \mathtt{nil})$ or $\mathtt{map}\ (\lambda x.\mathtt{s}\ x)$ for termination. Wanda indicates this by showing terms in functional notation; e.g., $\mathtt{map}(\lambda x.\mathtt{s}(x), \mathtt{cons}(\mathtt{0}, \mathtt{nil}))$.

▶ **Example 14.** Consider the toy system with $\mathcal{F} = \{\mathtt{a}, \mathtt{b} : \mathtt{o},\ \mathtt{f} : \mathtt{o} \to \mathtt{o} \to \mathtt{o},\ \mathtt{g} : \mathtt{o} \to (\mathtt{o} \to \mathtt{o}) \to \mathtt{o}\}$ and $\mathcal{R} = \{\ \mathtt{f}\ \mathtt{a}\ X \Rightarrow \mathtt{g}\ X\ (\mathtt{f}\ \mathtt{a}),\ \mathtt{g}\ \mathtt{a}\ F \Rightarrow F\ \mathtt{b}\}$. Then $minar(\mathtt{a}) = minar(\mathtt{b}) = 0$, $minar(\mathtt{g}) = 2$ and $minar(\mathtt{f}) = 1$ (since $\mathtt{f}$ occurs both with 1 or 2 arguments, we must choose the smaller value). Wanda prints these rules as $\mathtt{f}(\mathtt{a})\ X \Rightarrow \mathtt{g}(X,\ \mathtt{f}(\mathtt{a}))$ and $\mathtt{g}(\mathtt{a}, F) \Rightarrow F\ \mathtt{b}$.

We do *not* η-expand as part of uncurrying. To illustrate why not, note that the above system is terminating, but its η-long variant, which has a rule $\mathtt{f}\ \mathtt{a}\ X \Rightarrow \mathtt{g}\ (\lambda z.\mathtt{f}\ \mathtt{a}\ z)$, is not.

3 Non-termination

As Wanda's focus is on proving termination, the available non-termination techniques are currently quite minimal. There are three methods. The first two are very quick, and are applied at the start of the analysis, before termination is considered. The last one is employed when dependency pairs are initiated, as it is combined with the simplification given in §5.2.

3.1 Detecting obvious loops

An AFSM is clearly non-terminating if there is a reduction $s \Rightarrow^*_{\mathcal{R}} t$ such that $t \trianglerighteq s\gamma$ for some γ. To discover such loops, Wanda takes the left-hand side of a rule, replaces meta-variable applications $Z[x_1, \ldots, x_k]$ by variable applications $y\ x_1 \cdots x_k$, and performs a breadth-first search on reducts to see whether any instances of the original term appear, not going beyond the first 1000. If the `betafirst` runtime argument is given, then reducts are β-normalised before this test is done. This simple method will not find any sophisticated counterexamples for termination, but is quick and easy, and often catches mistakes in a recursive call.

In the future, it would be natural to extend this module to use semi-unification [31] instead of matching, as done for first-order rewriting in [26]. However, this would require the design of a higher-order semi-unification algorithm. Similarly, Wanda could be strengthened by creating higher-order variants of existing first-order non-termination techniques (e.g., [17, 45, 46]), but this would require substantial new work to develop the theory.

3.2 The $\omega\omega$ counterexample

Wanda also has one truly higher-order non-termination technique, which does not build on first-order methods. This technique recognises a particular kind of rule that leads to non-termination in a non-obvious way. The idea is to build a variation of the λ-term $\omega\omega$ in the untyped λ-calculus, where $\omega = \lambda x.xx$. Note that $\omega\omega$ reduces to itself in one $\Rightarrow_\beta$-step.

Let a *context* be a meta-term $\underline{C}[\Box_1, \ldots, \Box_n]$ containing n typed holes $\Box_i$, and denote $\underline{C}[s_1, \ldots, s_n]$ for the same meta-term with each $\Box_i$ replaced by s_i. Wanda identifies rules $\ell \Rightarrow r$ where ℓ has the form $\underline{C}[\underline{D}[Z], X]$ such that:

- $Z : \sigma_1 \to \ldots \to \sigma_n \to \tau \in \mathcal{M}$, where τ is the type of ℓ;
- there is some i with $X : (\sigma_i, 0) \in \mathcal{M}$ and also $\underline{D}[Z]$ has type σ_i;
- r can be written as $\underline{E}[Z\ s_1 \cdots s_{i-1}\ X\ s_{i+1} \cdots s_n]$
- X and Z do not appear at other positions in $\underline{C}$ or $\underline{D}$.

If this is satisfied, Wanda concludes non-termination with the following justification. Let γ be the substitution that maps each $Y : \pi_1 \to \ldots \to \pi_m \to \iota$ in the rule, aside from Z and X, to a term $\lambda x_1 \ldots x_m.y\ x_1 \cdots x_m$ (with y a variable), and let $\omega := \underline{D}\gamma[\lambda x_1 \ldots x_n.\underline{C}\gamma[x_i, x_i]]$. Let $\delta := \gamma \cup [X := \omega, Z := \lambda x_1 \ldots x_n.\underline{C}\gamma[x_i, x_i]]$. Then $\underline{C}\gamma[\omega, \omega] = \ell\delta \Rightarrow_{\mathcal{R}} \underline{E}[(\lambda x_1 \ldots x_n.\underline{C}\gamma[x_i, x_i])\ s_1 \cdots \omega \cdots s_n)]\delta \Rightarrow^*_\beta \underline{E}[\underline{C}\gamma[\omega, \omega]]\delta \trianglerighteq \underline{C}\gamma[\omega, \omega]$, a loop.

The method above is specialised for AFSMs that originate from AFSs (as used in the Termination Competition): it is designed for meta-variables that do not take any arguments. If meta-variables do take arguments, and for instance $\lambda x_1 \ldots x_n.Z[x_1, \ldots, x_n]$ is used instead of Z, we *probably* have a similar counter-example – depending on how Z and X are used in $\underline{E}$ (it is possible that $\underline{E}[]\delta$ does not contain any copies of $\Box_1$). Wanda tries to recognise such variations of the meta-variables, and tests whether the counterexample still applies.

3.3 Using a first-order tool

Finally, it is clear that an AFSM $(\mathcal{F}, \mathcal{R})$ is non-terminating if there is a subset $\mathcal{R}' \subseteq \mathcal{R}$ such that $\Rightarrow_{\mathcal{R}'}$ is non-terminating. An interesting subset is the set of rules that can be viewed as first-order (i.e., rules that do not use λ, that only use function symbols with a type declaration $\iota_1 \to \ldots \to \iota_m \to \iota_0$ with all $\iota_i \in \mathcal{S}$, and where function symbols only occur fully applied). This subset is easier to analyse, as known methods for first-order rewriting apply.

Thus, Wanda extracts this first-order part, to pass it to a dedicated first-order (non-) termination tool. The main problem with this approach is that existing tools do not consider types. This can make a difference, as shown by an example due to Toyama [50]:

▶ **Example 15.** Let $\mathcal{F} = \{\mathtt{0} : \mathtt{a},\ \mathtt{1} : \mathtt{a}, \mathtt{f} : \mathtt{a} \to \mathtt{a} \to \mathtt{a} \to \mathtt{a},\ \mathtt{g} : \mathtt{b} \to \mathtt{b} \to \mathtt{b}\}$ and $\mathcal{R} = \{\mathtt{f}(X, X, X) \Rightarrow \mathtt{f}(\mathtt{0}, \mathtt{1}, X),\ \mathtt{g}(X, Y) \Rightarrow X,\ \mathtt{g}(X, Y) \Rightarrow Y\}$. This system is terminating, because there is no term of type $\mathtt{a}$ that reduces to both $\mathtt{0}$ and $\mathtt{1}$. However, there is an *untypable* term that loops by these rules: $\mathtt{f}(\mathtt{0},\ \mathtt{1},\ \mathtt{g}(\mathtt{0}, \mathtt{1})) \Rightarrow_{\mathcal{R}} \mathtt{f}(\mathtt{g}(\mathtt{0}, \mathtt{1}),\ \mathtt{g}(\mathtt{0}, \mathtt{1}),\ \mathtt{g}(\mathtt{0}, \mathtt{1})) \Rightarrow_{\mathcal{R}} \mathtt{f}(\mathtt{0},\ \mathtt{g}(\mathtt{0}, \mathtt{1}),\ \mathtt{g}(\mathtt{0}, \mathtt{1})) \Rightarrow_{\mathcal{R}} \mathtt{f}(\mathtt{0},\ \mathtt{1},\ \mathtt{g}(\mathtt{0}, \mathtt{1}))$. Thus, a first-order termination tool (which does not consider types) would conclude non-termination.

Now, if the first-order subset is *orthogonal*, then it is terminating if and only if it is terminating without regarding types as observed in [20] (using a combination of results in [19] and [27]). Thus, in this case Wanda can use an arbitrary first-order tool without inhibitions. The same is true if the set of first-order rules uses only one sort. If neither of those cases holds, Wanda investigates the output of the first-order tool to see whether a non-terminating term is given, and if so, tests whether it is well-sorted.

Comment: unfortunately, the standard output format for the Termination Competition does not require tools to output a non-terminating term if `NO` is answered. Thus, any common first-order tool can be used if $\mathcal{R}'$ is orthogonal or has only one sort, but otherwise a specialised tool with the right output format is needed. For this, Wanda uses a custom adaptation of AProVE [24]. As AProVE is currently not open-source, this is not included in Wanda's release.

4 Orderings

At the heart of Wanda's termination techniques are *reduction pairs*. These are orderings on terms – generated by an ordering on meta-terms – which can be used both as part of the dependency pair framework (§5) and on their own to simplify a termination proof.

▶ **Definition 16.** *A* reduction pair *is a pair $(\succsim, \succ)$ of a quasi-ordering and a well-founded ordering on meta-terms of the same type, such that:*

- *$\succsim$ and $\succ$ are* compatible*: $\succ \cdot \succsim$ is included in $\succ$;*
- *$\succsim$ and $\succ$ are* meta-stable*: if $s \succsim t$ and γ is a substitution on domain $FMV(s) \cup FMV(t)$, then $s\gamma \succsim t\gamma$ (and similar for $\succ$);*
- *$\succsim$ is* monotonic*: if $s \succsim t$, then $s\ u \succsim t\ u$ and $u\ s \succsim u\ t$ and $\lambda x.s \succsim \lambda x.t$*
- *$\succsim$* contains beta*: $(\lambda x.s)\ t \succsim s[x := t]$ if s and t are terms.*

A reduction pair is strongly monotonic *if moreover $\succ$ is monotonic.*

Strongly monotonic reduction pairs can be used in *rule removal*: if $\ell \succsim r$ for some rules, and $\ell \succ r$ for the remainder, then the rules in the remainder cannot occur infinitely often in a reduction sequence, and thus can be "removed" (they no longer need to be considered for the termination argument). Reduction pairs are also used – without the strong monotonicity requirement – in the dependency pair framework. It *would* be possible to also include rule removal with strongly monotonic reduction pairs in the framework rather than using it as a separate step; however, using it as a separate step often gives simpler termination proofs, and makes it possible to assess the strength of these reduction pairs in isolation.

Wanda has two ways to generate reduction pairs: *weakly monotonic interpretations* and *recursive path orderings*. Both ideas extend first-order methods, and use *functional notation*. This is an extension of uncurrying, where the remaining applications are replaced by function application, as follows: in every rule, every subterm of the left- or right-hand side of the form $s\ t$ is replaced by $@^{\sigma,\tau}(s,t)$, where $s : \sigma \to \tau$. The set of all symbols $@^{\sigma,\tau}$ that are used in the rules is added to $\mathcal{F}$, and the corresponding rules $@^{\sigma,\tau}(\lambda x.Z[x], Y) \Rightarrow Z[Y]$ are added to $\mathcal{R}$.

▶ **Example 17.** The AFSM of Example 14 is functionalised by replacing $\mathtt{f}(\mathtt{a})\ X$ in the uncurried rules by $@^{o,o}(\mathtt{f}(\mathtt{a}), X)$ and $F\ \mathtt{b}$ by $@^{o,o}(F, \mathtt{b})$. Thus, we obtain the rules:

$$\begin{array}{rcl@{\qquad\qquad}rcl} @^{o,o}(\mathtt{f}(\mathtt{a}), X) & \Rightarrow & \mathtt{g}(X, \mathtt{f}(\mathtt{a})) & @^{o,o}(\lambda x.Z[x], Y) & \Rightarrow & Z[Y] \\ \mathtt{g}(\mathtt{a}, F) & \Rightarrow & @^{o,o}(F, \mathtt{b}) & & & \end{array}$$

4.1 Weakly monotonic algebras

The idea of van de Pol's *weakly monotonic algebras* [47] is to assign valuations which map all function symbols $\mathtt{f}$ of type σ to a *weakly monotonic functional* $\mathcal{J}_{\mathtt{f}}$: an element of $[\![\sigma]\!]$, where $[\![\iota]\!]$ is the set of natural numbers for a sort ι and $[\![\sigma \to \tau]\!]$ is the set of those functions from $[\![\sigma]\!]$ to $[\![\tau]\!]$ that are weakly monotonic (i.e., if $a, b \in [\![\sigma]\!]$ and $a \geq b$, then $f(a) \geq f(b)$ for $f \in [\![\sigma \to \tau]\!]$, where $\geq$ is a point-wise comparison). This induces a value on closed terms, which can be extended to a reduction pair, as explained below.

Given a meta-term s in functional notation and a function α which maps each variable $x : \sigma$ occurring freely in s to an element of $[\![\sigma]\!]$ and each meta-variable $Z : (\sigma_1 \to \ldots \to \sigma_k \to \tau, k)$ to an element of $[\![\sigma_1 \to \ldots \to \sigma_k \to \tau]\!]$, we let $[s]^{\mathcal{J}}_{\alpha}$ be recursively defined as follows:

$$\begin{array}{rcl@{\qquad}rcl} [x]^{\mathcal{J}}_{\alpha} & = & \alpha(x) & [\mathtt{f}(s_1, \ldots, s_k)]^{\mathcal{J}}_{\alpha} & = & \mathcal{J}_{\mathtt{f}}([s_1]^{\mathcal{J}}_{\alpha}, \ldots, [s_k]^{\mathcal{J}}_{\alpha}) \\ [\lambda x.s]^{\mathcal{J}}_{\alpha} & = & u \mapsto [s]^{\mathcal{J}}_{\alpha \cup [x:=u]} & [Z[s_1, \ldots, s_k]]^{\mathcal{J}}_{\alpha} & = & \alpha(Z)([s_1]^{\mathcal{J}}_{\alpha}, \ldots, [s_k]^{\mathcal{J}}_{\alpha}) \end{array}$$

(This follows the definition of $[\cdot]^{\mathcal{J}}_{\alpha}$ for functionalised AFSs in [21], but extends it with a case for meta-variable applications.) For closed meta-terms ℓ, r, let $\ell \succ r$ if $[\ell]^{\mathcal{J}}_{\alpha} > [r]^{\mathcal{J}}_{\alpha}$ for all α, and $\ell \succsim r$ if $[\ell]^{\mathcal{J}}_{\alpha} \geq [r]^{\mathcal{J}}_{\alpha}$ for all α. Then $(\succsim, \succ)$ is a reduction pair if the valuations $\mathcal{J}_{@^{\langle\sigma,\tau\rangle}}$ are chosen to have $\mathcal{J}_{@^{\langle\sigma,\tau\rangle}}(F, X) \geq F(X)$. It is a strongly monotonic pair if each function $\mathcal{J}_{\mathtt{f}}$ (including each $\mathcal{J}_{@^{\langle\sigma,\tau\rangle}}$) is monotonic over $>$ in the first $\mathit{minar}(\mathtt{f})$ arguments.

In [21], a strategy is discussed to find interpretations based on *higher-order polynomials* for AFSs, and an automation using encodings of the ordering requirements into SAT. Wanda implements this methodology, only slightly adapted to take meta-variables into account.

▶ **Example 18.** We consider $\mathcal{R}_2$ in Example 6. Let $\mathcal{J}_{\mathtt{nil}} = 0$ and $\mathcal{J}_{\mathtt{cons}} = (n, m) \mapsto n + m + 1$ and $\mathcal{J}_{\mathtt{map}} = (f, n) \mapsto nf(n) + 2n + f(0)$ and $\mathcal{J}_{@^{\mathtt{nat},\mathtt{nat}}} = (f, n) \mapsto f(n) + n$. Then, writing $F := \alpha(Z),\ n := \alpha(H),\ m := \alpha(T)$, we have:

- $[\mathtt{map}(Z, \mathtt{nil})]^{\mathcal{J}}_{\alpha} = F(0) \geq 0 = [\mathtt{nil}]^{\mathcal{J}}_{\alpha}$
- $[\mathtt{map}(Z, \mathtt{cons}(H, T))]^{\mathcal{J}}_{\alpha} = (n + m + 1) \cdot F(n + m + 1) + 2 \cdot (n + m + 1) + F(0) > (F(n) + n) + (m \cdot F(m) + 2 \cdot m + F(0)) + 1 = [\mathtt{cons}(@^{\langle\mathtt{nat},\mathtt{nat}\rangle}(Z, H), \mathtt{map}(Z, T))]^{\mathcal{J}}_{\alpha}$
- $[@^{\mathtt{nat},\mathtt{nat}}(\lambda x.Z[x], H)]^{\mathcal{J}}_{\alpha} = F(H) + H \geq F(H) = [F[H]]^{\mathcal{J}}_{\alpha}$.

4.2 StarHorpo

The recursive path ordering [14] is a syntactic method to extend an ordering on function symbols to an ordering on first-order terms. There are various extensions (e.g. [18, 30]) including several higher-order variations (e.g. [7, 29]). However, these are mostly designed for rewriting with plain matching, and adapting them to work well with meta-variables is

non-trivial. Instead, Wanda uses a specialised definition, built using the same ideas as [29] but using *iterative* path orderings [33, 36] as a starting point. This is discussed in detail in [35, Ch. 5]; here, we note only the end result: a reduction pair that can be used on functionalised AFSMs and (unlike other higher-order recursive path orderings) is natively transitive.

Following [33, 36], StarHorpo employs a star mark $\star$ to indicate an *intent to decrease*; practically, $\mathtt{f}^\star_\sigma(s_1,\ldots,s_k)$ should be seen as an upper bound for all functional meta-terms of type σ which are *strictly smaller* than $\mathtt{f}(s_1,\ldots,s_k)$. Let $s^\star$ denote $\lambda x_1 \ldots x_n.\mathtt{f}^\star_\sigma(s_1,\ldots,s_k)$ if $s = \lambda x_1 \ldots x_n.\mathtt{f}(s_1,\ldots,s_k)$. If s has any other form, then $s^\star$ is undefined.

StarHorpo assumes given a *precedence* $\trianglerighteq$: a quasi-ordering on all symbols, whose strict part $\triangleright$ is well-founded; we let $\approx$ denote the equivalence relation $\trianglerighteq \cap \trianglelefteq$. We assume that there is a special symbol $\bot_\sigma$ for each type σ, which is minimal for $\trianglerighteq$ (i.e., $\mathtt{f} \trianglerighteq \bot_\sigma$ for all $\mathtt{f}$); $\bot^\star_\sigma$ is undefined. All symbols are assigned a *status* in $\{Lex, Mul\}$, such that $status(\mathtt{f}) = status(\mathtt{g})$ whenever $\mathtt{f} \approx \mathtt{g}$. Let $\succ^{\mathtt{f}}_\star$ denote either the lexicographic or multiset extension of $\succ_\star$, depending on the status of $\mathtt{f}$. Now the reduction pair $(\succeq_\star, \succ_\star)$ is given by the rules in Figure 1.

$(\succ)$	s	$\succ_\star$	t	if	$s^\star \succeq_\star t$	
(Var)	x	$\succeq_\star$	x	if	$x \in \mathcal{V}$	
(Abs)	$\lambda x.s$	$\succeq_\star$	$\lambda x.t$	if	$s \succeq_\star t$	
(Meta)	$Z[s_1,\ldots,s_k]$	$\succeq_\star$	$Z[t_1,\ldots,t_k]$	if	each $s_i \succeq_\star t_i$	
(Fun)	$\mathtt{f}(s_1,\ldots,s_n)$	$\succeq_\star$	$\mathtt{g}(t_1,\ldots,t_k)$	if	$\mathtt{f} \approx \mathtt{g}$ and $[s_1,\ldots,s_n] \succeq^{\mathtt{f}}_\star [t_1,\ldots,t_k]$	
(Put)	$\mathtt{f}(s_1,\ldots,s_n)$	$\succeq_\star$	t	if	$\mathtt{f}^\star_\sigma(s_1,\ldots,s_n) \succeq_\star t$ (for $\mathtt{f}(\vec{s}) : \sigma$)	
(Select)	$\mathtt{f}^\star_\sigma(s_1,\ldots,s_n)$	$\succeq_\star$	t	if	$s_i\langle \mathtt{f}^\star_{\tau_1}(\vec{s}),\ldots,\mathtt{f}^\star_{\tau_j}(\vec{s})\rangle \succeq_\star t$ (**) where $s_i : \tau_1 \to \ldots \to \tau_j \to \sigma$	(**)
(FAbs)	$\mathtt{f}^\star_{\sigma\to\tau}(s_1,\ldots,s_n)$	$\succeq_\star$	$\lambda x.t$	if	$\mathtt{f}^\star_\tau(s_1,\ldots,s_n,x) \succeq_\star t$	
(Copy)	$\mathtt{f}^\star_\sigma(s_1,\ldots,s_n)$	$\succeq_\star$	$\mathtt{g}(t_1,\ldots,t_k)$	if	$\mathtt{f} \triangleright \mathtt{g}$ and $\mathtt{f}^\star_{\tau_i}(\vec{s}) \succeq_\star t_i$ for $1 \le i \le k$	
(Stat)	$\mathtt{f}^\star_\sigma(s_1,\ldots,s_n)$	$\succeq_\star$	$\mathtt{g}(t_1,\ldots,t_k)$	if	$\mathtt{f} \approx \mathtt{g}$ and $\mathtt{f}^\star_{\tau_i}(\vec{s}) \succeq_\star t_i$ for $1 \le i \le k$ and $[s_1,\ldots,s_n] \succ^{\mathtt{f}}_\star [t_1,\ldots,t_k]$	
(Bot)	s	$\succeq_\star$	$\bot_\sigma$	if	$s : \sigma$	

The notation $s\langle t_1,\ldots,t_n\rangle$ applies s to $t_1,\ldots,t_n$ in the following sense: $s\langle\rangle = s$ and $(\lambda x.s)\langle t,\vec{u}\rangle = s[x := t]\langle\vec{u}\rangle$ and $\mathtt{f}(\vec{s})\langle t,\vec{u}\rangle = \mathtt{f}^\star_\tau(\vec{s},t)\langle\vec{u}\rangle$ and also $\mathtt{f}^\star_{\sigma\to\tau}(\vec{s})\langle t,\vec{u}\rangle = \mathtt{f}^\star_\tau(\vec{s},t)\langle\vec{u}\rangle$.

Figure 1 Rules of StarHorpo.

Note that $\succeq_\star$ and $\succ_\star$ only compare terms of the same type, and that marked symbols $\mathtt{f}^\star$ may occur with different types (indicated as subscripts) within a term. Symbols $\mathtt{f}^\star$ may also have varying numbers of arguments, but must always have at least $minar(\mathtt{f})$.

▶ **Example 19.** Given a function symbol $@ : (\sigma \to \tau) \to \sigma \to \tau$ (with σ and τ arbitrary types), we can prove $@(\lambda x.Z[x], Y) \succ Z[Y]$ as follows:

- by $(\succ)$, because $@^\star_\tau(\lambda x.Z[x], Y) \succeq_\star Z[Y]$
- by (Select), because $Z[@^\star_\sigma(\lambda x.Z[x], Y)] \succeq_\star Z[Y]$
- by (Meta), because $@^\star_\sigma(\lambda x.Z[x], Y) \succeq_\star Y$
- by (Select) because $Y \succeq_\star Y$ by (Meta).

Wanda uses StarHorpo in combination with *argument functions*: each function symbol $\mathtt{f}$ with $minar(\mathtt{f}) = k$ is mapped to a functionalised term $\lambda x_1 \ldots x_k.s$, and in a given functionalised meta-term, all occurrences of $\mathtt{f}(t_1,\ldots,t_k)$ are replaced by $s[x_1 := t_1,\ldots,x_k := t_k]$. If the reduction pair is required to be strongly monotonic (as is the case for rule removal), then $FV(s)$ must be $\{x_1,\ldots,x_k\}$. Argument functions are a generalisation of *argument filterings* [39], and were introduced in [37]. In Wanda, they are not restricted to being used

with dependency pairs (unlike [39, 37]), and s is limited to one of three forms: (1) x_i, (2) $\mathtt{f}'(x_{i_1}, \ldots, x_{i_n})$ (with $n \leq k$, all x_{i_j} distinct), or (3) $\bot_\sigma$. This effectively extends argument filterings with argument permutations and a mapping to one of the minimal constants $\bot_\sigma$.

Wanda combines the search for a suitable precedence and status function with the search for an argument function, using a SAT encoding following [35, Chapter 8.6].

▶ **Example 20.** Consider the (first-order) AFSM with just one sort $\mathtt{o}$ and the following rules:

$$\mathtt{f}\ X\ (\mathtt{s}\ Y) \quad \Rightarrow \quad \mathtt{g}\ Y\ (\mathtt{s}\ (\mathtt{s}\ X)) \qquad \mathtt{f}\ X\ Y \quad \Rightarrow \quad \mathtt{g}\ \mathtt{a}\ \mathtt{b} \qquad \mathtt{g}\ X\ (\mathtt{s}\ Y) \quad \Rightarrow \quad \mathtt{f}\ Y\ X$$

Then $\mathit{minar}(\mathtt{f}) = \mathit{minar}(\mathtt{g}) = 2$. We use the argument functions $\pi(\mathtt{f}) = \lambda x.\lambda y.\mathtt{f}'(y, x)$ and $\pi(\mathtt{g}) = \lambda x.\lambda y.\mathtt{g}'(x, y)$ and $\pi(\mathtt{s}) = \lambda x.\mathtt{s}'(x)$ and $\pi(\mathtt{a}) = \pi(\mathtt{b}) = \bot_{\mathtt{nat}}$ to get the requirements:

$$\begin{array}{rcl@{\qquad}rcl} \mathtt{f}'(\mathtt{s}'(Y), X) & \succ & \mathtt{g}'(Y, \mathtt{s}'(\mathtt{s}'(X))) & \mathtt{f}'(Y, X) & \succsim & \mathtt{g}'(\bot_{\mathtt{nat}}, \bot_{\mathtt{nat}}) \\ & & & \mathtt{g}'(X, \mathtt{s}'(Y)) & \succ & \mathtt{f}'(X, Y) \end{array}$$

This is easily handled with $\mathtt{f}' \approx \mathtt{g}' \blacktriangleright \mathtt{s}'$, and $\mathit{status}(\mathtt{f}') = \mathit{status}(\mathtt{g}') = \mathit{Lex}$. This example relies on $\mathtt{a}$ and $\mathtt{b}$ being mapped to $\bot_{\mathtt{nat}}$. Such use of a minimal constant originates in [48].

5 Dependency Pairs

After trying to prove non-termination using the methods in §3.1–3.2, and removing as many rules as possible with strongly monotonic reduction pairs, control is passed to the *dependency pair (DP) framework*. Like the first-order DP framework [25], this is an extendable framework for termination (and non-termination), which new termination methods can easily be plugged into in the form of "processors". This framework encompasses all remaining termination techniques, but does not currently contain any processors for non-termination. The DP framework is detailed in [37, 22] and [35, Ch. 6–7]. Let us here consider a high-level overview.

5.1 The DP framework

The relatively simple form of the DP framework in Wanda operates on pairs $(\mathcal{P}, \mathcal{R})$ called *DP problems*. For a given AFSM, an initial pair is generated, which must be proved "finite" (also called "non-looping" in [2, 37]). If this property applies, then the AFSM is terminating.

Now, a *processor* is a function that maps a DP problem ρ to a finite set of DP problems. Wanda has a list of processors M such that ρ is finite if and only if all elements of $M(\rho)$ are finite; moreover, either $M(\rho) = \{\rho\}$, or all elements of $M(\rho)$ are strictly smaller than ρ (counting the number of elements in $\mathcal{P}$ and $\mathcal{R}$). Wanda then applies the following algorithm:

1. Let A be the set containing just the initial DP problem $(\mathcal{P}, \mathcal{R})$.
2. If $A = \emptyset$ then return `YES`.
3. Otherwise, choose an arbitrary element $\rho \in A$.
4. Find the first processor M in the list of processors such that $M(\rho) \neq \{\rho\}$.
5. If such a processor cannot be found, then the process has failed; return `MAYBE`.
6. Otherwise, let $A := (A \setminus \{\rho\}) \cup M(\rho)$, and go back to Item 2.

Note that, throughout the process, we retain the following property: the original AFSM is terminating if the initial DP problem is finite, which holds if and only if all elements in A are finite. This is why the conclusion in Item 2 is correct.

The processors used are, in order: the dependency graph, the subterm criterion, the computable subterm criterion, formative rules, and reduction pairs with usable rules (first polynomial interpretations, then `StarHorpo`). All processors are explained in [35, 22].

5.2 Delegation to a first-order prover

Following [20], the framework starts (as part of Item 1 in §5.1) by identifying the *first-order* rules in the AFSM. These are functionalised and passed to an external first-order termination tool; if the full AFSM is not orthogonal then additionally all rules in $\mathcal{C}_\epsilon = \{\mathtt{c}_\iota(X,Y) \Rightarrow X,\ \mathtt{c}_\iota(X,Y) \Rightarrow Y \mid \iota \in \mathcal{S}\}$ are added (with $\mathtt{c}_\iota : \iota \to \iota \to \iota$ fresh function symbols).[1]

If the tool detects termination, then this is stored, as it allows all dependency pairs for these first-order rules to be omitted from the set of generated DPs. If the tool returns `NO` and no $\mathcal{C}_\epsilon$ rules were added, then non-termination is concluded as explained in §3.3. Otherwise, the remaining cases of §3.3 are tested with a dedicated non-termination prover.

5.3 Static and Dynamic DPs

To complete item 1 – so to generate the initial DP problem $(\mathcal{P},\mathcal{R})$ – there are two different approaches, originating from distinct lines of work around the same period [37, 38]. In both cases, an AFSM $(\mathcal{F},\mathcal{R})$ gives an initial DP problem $(\bigcup\{DP(\rho) \mid \rho \in \mathcal{R}\},\ \mathcal{R} \cup OptionalExtra)$, where the set $DP(\rho)$ of dependency pairs generated for a given rule varies between the two approaches. In both cases, the elements of $DP(\rho)$ with ρ a first-order rule may be omitted if the first-order part was proved terminating following §5.2. Unlike the name suggests (as this differs from the first-order definition), these dependency pairs are actually *triples* of a pattern of the form $\mathtt{f}\ \ell_1 \dots \ell_n$, a meta-term r and a set; this is discussed in more detail in [35, 22].

In the *dynamic* approach, each $DDP(\rho)$ contains triples whose second component r has a form $\mathtt{g}\ r_1 \cdots r_m$ or $Z[r_1,\dots,r_m]$; the latter kind is called a "collapsing" DP. In the *static* approach, $SDP(\rho)$ contains no collapsing DPs, but may have DPs where $FMV(r) \not\subseteq FMV(\ell)$. Both fresh meta-variables in r and collapsing DPs are complications not present in the first-order setting, which make some of the processors weaker. The static approach for generating DPs can only be used if some restrictions on the AFSM are satisfied, but when applicable often gives an easier termination proof than the dynamic one.

The notion of a finite problem and the processors used in Wanda can all be defined generally enough to apply for both the static and dynamic approach. Hence, once the initial DP problem is generated, the same DP framework can be used for both. Wanda tries dynamic DPs first, and if this fails, falls back to static DPs. However, if $\bigcup\{SDP(\rho) \mid \rho \in \mathcal{R}\} \subseteq \bigcup\{DDP(\rho) \mid \rho \in \mathcal{R}\}$, this first step is omitted and only the static approach is tried.

▶ **Example 21.** For $\mathcal{R}_1$ in Example 6, the dynamic approach generates $(\{(1),(2)\},\mathcal{R}_1)$ with:

$$\begin{array}{lllll} (1) & \mathtt{map}^\sharp\ (\lambda x.Z[x])\ (\mathtt{cons}\ H\ T) & \Rrightarrow & \mathtt{map}^\sharp\ (\lambda x.Z[x])\ T & (\emptyset) \\ (2) & \mathtt{map}^\sharp\ (\lambda x.Z[x])\ (\mathtt{cons}\ H\ T) & \Rrightarrow & Z[H] & (\emptyset) \end{array}$$

The static approach generates $(\{(1)\},\mathcal{R}_1)$. Thus, Wanda does not try the dynamic approach.

6 Experimental results

To test the power of both Wanda as a whole, and individual techniques, various configurations of Wanda were tested on two data sets: (1) the "higher order union beta" benchmarks in the Termination Problem DataBase [13] (which are used in the International Termination

[1] These rules allow for the construction of a term that can be reduced to all elements of an arbitrary finite set of terms with the same type. They are trivially discarded by many termination techniques, but may complicate analysis because they turn the system non-confluent.

	YES	NO	MAYBE	TIMEOUT	Avg. time
Full	188	16	25	32	1.14
Only rule removal	123	0	118	20	1.13
Only `StarHorpo`	111	0	141	9	0.24
Only interpretations	59	0	156	46	0.07
Only dependency pairs	186	0	42	33	1.02
only static DPs	152	0	86	23	0.55
only dynamic DPs	167	0	58	36	1.30
no first-order tool	183	9	47	22	0.90
no overestimation	155	16	25	65	0.75

Figure 2 Experimental results on the TPDB (261 benchmarks).

Competition [12]), and (2) the pattern HRSs in the COPS (Confluence Problems) database [10] (which are used in the International Confluence Competition [11]), most of which were translated to AFSMs by the tool CSI^ho [42]. Wanda was executed with a timeout of 60 seconds, on a Lenovo Thinkpad T420, using AProVE [24] as a first-order termination prover, and MiniSAT [16] as a SAT-solver. The results are discussed below. Note that the average time only takes `YES` and `NO` results into account; in particular, `TIMEOUT`s are not considered.

An evaluation page with detailed results is available at:

`https://www.cs.ru.nl/~cynthiakop/experiments/fscd20/`

6.1 Benchmarks from the TPDB

The results on the termination problem database are given in Figure 2. The first test is Wanda's default behaviour, the next three use only rule removal (with both techniques or only one), and the next three use only the DP framework (either full or with only one way of generating the initial DP problem). The final tests disable specific features in the full version: using a first-order termination tool, and overestimating AFSs as described in §2.2. The longest successful evaluation is 20.46 seconds, so not close to the 60 second timeout.

The tests show that rule removal is not as effective as dependency pairs, but does help a little: when it is disabled, Wanda loses two benchmarks (and does not gain any). This could be avoided by implementing rule removal as a processor in the DP framework, but this has thus far not been done (the implementation is not entirely straightforward due to the different requirements imposed by the DP framework). The effect of rule removal on speed is variable: rule removal often *succeeds* fast, but may take a long time to *fail*. Thus, when both are tried, the solution speed could go either way. Within rule removal, `StarHorpo` is much more powerful than polynomial interpretations, but the techniques are incomparable: there are 12 benchmarks that can be handled by interpretations but not `StarHorpo`.

Also the two styles of dependency pairs are incomparable: the dynamic approach seems to give a bit more power, but there are benchmarks that can be handled with static DPs and not with dynamic ones. Moreover, the static approach is significantly faster.

Worth noting is that there are 16 benchmarks Wanda can prove non-terminating, of which 7 are found by AProVE. Of the remainder, manual checking shows that 7 have obvious loops, and 2 admit the $\omega\omega$ example. For termination, using AProVE gives a modest gain (five benchmarks). The last row deserves some further discussion. Due to unclear documentation on the competition's format, the 85 newest benchmarks in this database are all "fake HRS": like the system in Example 11, the left-hand sides often have subterms such as $F\ x\ y$ where

	YES	NO	MAYBE	TIMEOUT	Avg. time
Full	43	30	19	1	0.09
Only rule removal	37	0	56	0	0.11
Only `StarHorpo`	33	0	60	0	0.17
Only interpretations	21	0	72	0	0.01
Only dependency pairs	43	0	49	1	0.51
only static DPs	37	0	56	0	0.26
only dynamic DPs	40	0	52	1	0.77
no first-order tool	43	30	19	1	0.07

Figure 3 Experimental results on the COPS database (93 benchmarks).

F is a free variable. Wanda spends more time on these benchmarks than others, since not only the true translation to AFSM is considered, but also an overestimation that is often easier to handle. When overestimating is disabled, Wanda is faster, but significantly weaker.

The first-order tool. It is worth noting that more than fifty benchmarks in this database are actually first-order systems with one or two (typically trivial) higher-order rules. Indeed, about 25 of Wanda's `TIMEOUT`s are due to AProVE timing out on a complicated first-order fragment. This raises the question whether the choice of first-order tool is significant.

The answer is ambiguous. For *non-termination*, Wanda relies on an explicit counter-example, which only the customised version of AProVE provides; without it, Wanda loses 7 `NO`s. For *termination*, comparing Wanda's performance when instead coupled with NaTT or MU-TERM, we found that NaTT outperforms both AProVE and MU-TERM by 13 benchmarks. However, this advantage is local: the "higher-order union beta" category of the TPDB has seven sub-directories, each representing a batch of (often similar) benchmarks that were added at the same time. On six of those seven, Wanda performs almost identically whichever first-order tool is used: MU-TERM and AProVE give one benchmark that NaTT fails, and all other answers are the same. In the seventh, NaTT wins 14 benchmarks over the others.

Looking at all benchmarks, we observe: the *only* cases where using a first-order tool helps, are combinations of a challenging first-order TRS and a quite simple higher-order part: it can be handled with static DPs and one of the subterm criterion processors [22]. Which first-order tool is the best for the job depends only on the form of the first-order part.

6.2 Benchmarks from COPS

Figure 3 shows the experimental results on AFSMs translated from the Confluence Problems database (COPS) [10]. Here, unlike the benchmarks from the TPDB, meta-variables are used with arguments. Even so, the comparative results between rule removal and full Wanda, and between static, dynamic and full dependency pairs, are similar to the TPDB results. There are relatively far more `NO` answers, which seems to be because COPS contains more non-terminating systems (and quite a few trivially so). This is explained by the purpose of the database: confluence is harder to prove for non-terminating than terminating systems.

7 Practical use

Wanda is designed to run on a Linux terminal, and is invoked by supplying one or more input files, and zero or more runtime parameters that customise the behaviour. Runtime parameters range from purely aesthetical commands (e.g., to indicate that Wanda should

use coloured output), to commands that make Wanda output properties of the given system (e.g., to indicate whether a system has η-long form) or that modify the termination checking behaviour (e.g., the previously mentioned `betafirst` parameter). Wanda has a fixed strategy – that is, techniques are always applied in the same order – but certain techniques can be disabled for practical experiments; this was done in §6. The full range of parameters is documented in the `README.txt` file included with the distribution. Some pertinent commands are:

- `-d` ⟨`methods`⟩ disables the given methods; for example, use `./wanda.exe -d nt,poly,dp` to disable non-termination analysis, algebra interpretations and dependency pairs; this forces Wanda to generate a proof using `StarHorpo`, if one can be found;
- `-i` ⟨`tool`⟩ tells Wanda to use the given first-order termination tool as back-end, which must be located in the `resources/` sub-directory. If not given, Wanda uses the file "firstorder-prover". Similarly, `-n` ⟨`tool`⟩ tells Wanda to use the given tool for *non-termination* analysis when this is done in a separate step.

In standard usage, Wanda takes an input file describing an AFSM or AFS, performs an analysis following §3–5 and then prints `YES` (a termination proof was found), `NO` (a non-termination proof was found) or `MAYBE` (neither could be proved). In the first two cases, this is followed by a human-readable proof. If more than one input file is supplied, Wanda prints the name of each file, followed by the answer and possibly proof. A timeout may be supplied (following the standard for the termination competition) but is ignored.

8 Conclusions and directions for future work

This paper has discussed the various techniques used in Wanda, and how they are applied. Wanda is only one of several higher-order tools, and interestingly, *incomparable* to others: there are benchmarks that Wanda can handle and other tools cannot, and vice versa. This is because all tools that have participated in the Termination Competition have focused on different techniques. For Wanda, the main termination approach is the DP framework.

There are many directions for improvement. Most pertinently, due to the presence of a large database of termination benchmarks in the competition format [13], Wanda has been optimised for AFSs and is decidedly weak in the presence of meta-variables with arguments. Moreover, non-termination analysis is very limited and does not take advantage of the DP framework. Other improvements could be to further extend first-order termination techniques, to build on primarily higher-order techniques like sized types [5], and to support AFSMs with polymorphic types. Automatic certification as has been done for first-order rewriting [49] would be a highly interesting direction to pursue, but would require a vast amount of work to build up the formalisation library. Finally, Wanda's usability could be substantially improved by the addition of a web interface, for example using the EasyInterface toolkit [15].

A complete discussion of most techniques in Wanda and the technology behind automating them is available in the author's PhD thesis [35]. Wanda is open-source and available from `http://wandahot.sourceforge.net/`. The snapshot that was used in the present paper (including both open- and closed-source back-ends) is available from the evaluation pages:

`https://www.cs.ru.nl/~cynthiakop/experiments/fscd20/`

References

1 P. Aczel. A general Church-Rosser theorem, 1978. Unpublished Manuscript, University of Manchester. http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGeneralChurch-RosserTheorem.pdf.

2 T. Aoto and Y. Yamada. Argument filterings and usable rules for simply typed dependency pairs. In *Proceedings of FroCoS*, volume 5749 of *LNAI*, pages 117–132. Springer, 2009.

3 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

4 F. Blanqui. HOT – an automated termination prover for higher-order rewriting. URL: http://rewriting.gforge.inria.fr/hot.html.

5 F. Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In *Proceedings of RTA*, volume 3091 of *LNCS*, pages 24–39. Springer, 2004.

6 F. Blanqui, J. Jouannaud, and M. Okada. Inductive-data-type systems. *Theoretical Computer Science*, 272(1-2):41–68, 2002.

7 F. Blanqui, J. Jouannaud, and A. Rubio. The computability path ordering: The end of a quest. In *Proceedings of CSL*, volume 5213 of *LNCS*, pages 1–14. Springer, 2008.

8 C. Borralleras and A. Rubio. THOR – an automatic tool for proving termination of higher-order rewriting. URL: https://www.cs.upc.edu/~albert/term.html.

9 C. Borralleras and A. Rubio. A monotonic higher-order semantic path ordering. In *Proceedings of LPAR*, volume 2250 of *LNAI*, pages 531–547. Springer, 2001.

10 Community. Confluence Problems (COPS). URL: https://cops.uibk.ac.at/?q=prs.

11 Community. The international Confluence Competition (CoCo). URL: http://coco.nue.riec.tohoku.ac.jp/.

12 Community. Termination Portal. URL: http://www.termination-portal.org/wiki/Termination_Competition.

13 Community. Termination Problem DataBase (TPDB). URL: http://termination-portal.org/wiki/TPDB.

14 N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

15 J. Doménech, S. Genaim, E.B. Johnsen, and R. Schlatte. EasyInterface: A toolkit for rapid development of GUIs for research prototype tools. In *Proceedings of FASE*, volume 10202 of *LNCS*, pages 379–383. Springer, 2017.

16 N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004. See also http://minisat.se/.

17 F. Emmes, T. Enger, and J. Giesl. Proving non-looping non-termination automatically. In *Proceedings of IJCAR*, volume 7364 of *LNAI*, pages 225–240. Springer, 2012.

18 M. Ferreira and H. Zantema. Syntactical analysis of total termination. In *Proceedings of ALP*, volume 850 of *LNCS*, pages 204–222. Springer, 1994.

19 C. Fuhs, J. Giesl, M. Parting, P. Schneider-Kamp, and S. Swiderski. Proving termination by dependency pairs and inductive theorem proving. *Journal of Automated Reasoning*, 47(2):133–160, 2011.

20 C. Fuhs and C. Kop. Harnessing first order termination provers using higher order dependency pairs. In *Proceedings of FroCoS*, volume 6989 of *LNAI*, pages 147–162. Springer, 2011.

21 C. Fuhs and C. Kop. Polynomial interpretations for higher-order rewriting. In *Proceedings of RTA*, volume 15 of *LIPIcs*, pages 176–192. Dagstuhl, 2012.

22 C. Fuhs and C. Kop. A static higher-order dependency pair framework. In *Proceedings of ESOP*, volume 11423 of *LNCS*, pages 752–782, 2019.

23 G. Genestier. SizeChangeTool: A termination checker for rewriting dependent types. In *Proceedings of HOR*, pages 14–19, 2019. URL: https://hal.archives-ouvertes.fr/hal-02442465/document.

24 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.

25 J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proceedings of LPAR*, volume 3452 of *LNAI*, pages 301–331. Springer, 2005.

26 J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proceedings of FroCoS*, volume 3717 of *LNAI*, pages 216–231. Springer, 2005.

27 B. Gramlich. Abstract relations between restricted termination and confluence properties of rewrite systems. *Fundamenta Informaticae*, 24(1-2):3–23, 1995.

28 M. Hamana. PolySOL – an automatic tool for confluence and termination of polymorphic second-order systems. URL: `http://www.cs.gunma-u.ac.jp/hamana/polysol/`.

29 J. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proceedings of LICS*, IEEE, pages 402–411, 1999.

30 S. Kamin and J.-J. Lévy. Two generalizations of the recursive path ordering, 1980. Unpublished Manuscript, University of Illinois.

31 D. Kapur, P. Musser, D. Narendran, and J. Stillman. Semi-unification. In *Proceedings of FSTTCS*, volume 338 of *LNCS*, pages 435–454, 1988.

32 J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1-2):279–308, 1993.

33 J.W. Klop, V. van Oostrom, and R. de Vrijer. Iterative lexicographic path orders. In *Essays dedicated to Joseph A. Goguen on the Occasion of his 65th Birthday*, volume 4060 of *LNCS*, pages 541–554. Springer, 2006. Festschrift.

34 C. Kop. Simplifying algebraic functional systems. In *Proceedings of CAI*, volume 6742 of *LNCS*, pages 201–215. Springer, 2011.

35 C. Kop. *Higher Order Termination*. PhD thesis, VU University Amsterdam, 2012.

36 C. Kop and F. van Raamsdonk. A higher-order iterative path ordering. In *Proceedings of LPAR*, volume 5330 of *LNAI*, pages 697–711, 2008.

37 C. Kop and F. van Raamsdonk. Dynamic dependency pairs for algebraic functional systems. *Logical Methods in Computer Science*, 8(2):10:1–10:51, 2012. Special Issue for RTA '11.

38 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.

39 K. Kusakari, M. Nakamura, and Y. Toyama. Argument filtering transformation. In *Proceedings of PPDP*, volume 1702 of *LNCS*, pages 47–61. Springer, 1999.

40 S. Lucas, C. Marché, and J. Meseguer. Operational termination of conditional term rewriting systems. *Information Processing letters*, 95(4):446–453, 2005.

41 D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

42 J. Nagele. CoCo 2016 participant: CSI^ho 0.2. ; tool webpage: `http://cl-informatik.uibk.ac.at/software/csi/ho/`. URL: `http://coco.nue.riec.tohoku.ac.jp/2016/papers/csiho.pdf`.

43 T. Nipkow. Higher-order critical pairs. In *Proceedings of LICS*, pages 342–349. IEEE, 1991.

44 K. Onozawa, K. Kikuchi, T. Aoto, and Y. Toyama. ACPH: System description for CoCo 2016. URL: `http://coco.nue.riec.tohoku.ac.jp/2016/papers/acph.pdf`.

45 É. Payet. Loop detection in term rewriting using the eliminating unfoldings. *Theoretical Computer Science*, 403(2-3):307–327, 2008.

46 É. Payet. Guided unfoldings for finding loops in standard term rewriting. In *Proceedings of LOPSTR*, volume 11408 of *LNCS*, pages 22–37, 2018.

47 J.C. van de Pol. *Termination of Higher-order Rewrite Systems*. PhD thesis, University of Utrecht, 1996.

48 R. Thiemann, G. Allais, and J. Nagele. On the formalization of termination techniques based on multiset orderings. In *Proceedings of RTA*, volume 15 of *LIPIcs*, pages 339–354. Dagstuhl, 2012.

49 R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proceedings of TPHOLs*, volume 5674 of *LNCS*, pages 452–468. Springer, 2009.

50 Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25(1):141–143, 1987.

51 H. Zantema. Termination of context-sensitive rewriting. In *Proceedings of RTA*, volume 1232 of *LNCS*, pages 172–186, 1997.