AUTOMATIC TERMINATION ANALYSIS USING WANDA

CYNTHIA KOP

Department of Software Science, Radboud University Nijmegen
e-mail address: 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 WANDA's underlying methodology. Most pertinently, this includes a higher-order dependency pair framework, weakly monotonic algebras through higher-order polynomials, and a variation of the higher-order recursive path ordering. All techniques are employed automatically using SAT encodings.

1. Introduction

Termination of term rewriting systems has been an area of active research for several decades. In recent years the field of *automatically* proving termination has flourished, and several strong provers have been developed to participate against each other in the annual *International Termination Competition* [27].

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, the higher-order category of the termination category has seen the participation of a range of tools (HOT [2]), THOR [6], WANDA), each using different techniques.

WANDA, a tool structured primarily around *higher-order dependency pairs*, has been the leading tool in this category since 2013. WANDA was also the tool of choice as a termination back-end in the higher-order category of the *International Confluence Competition* [7], with both participating tools in 2016 (ACPH [25] and CSIˆho [23]) delegating termination questions to WANDA.

In this paper we will discuss the most important techniques used in WANDA. To this end we follow roughly the structure of an analysis by WANDA: first a higher-order TRS is read and (if necessary) translated into WANDA's internal formalism, *AFSMs* (§2); then basic techniques for non-termination (§3) and for simple termination proofs using reduction pairs (§4) are applied. Finally, responsibility is passed to the dependency pair framework (§5).

2. 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. To support (non-)termination proofs in several popular formalisms at once, WANDA uses her own internal format, *Algebraic Functional Systems with*

Meta-variables. AFSMs are essentially simply-typed CRSs [16] and also largely correspond to the formalism in [4]; they are fully explained in [19, Ch. 2] and in [10]. We here present an overview which assumes familiarity with term rewriting and simply-typed lambda-calculus.

Terms are built from a set of simply-typed variables $\mathcal{V}$ and a set $\mathcal{F}$ of simply-typed function symbols, using abstraction and application to form well-typed expressions. *Meta-terms* are built from $\mathcal{V}$, $\mathcal{F}$ and a set $\mathcal{M}$ of meta-variables, each equipped with a type declaration $[\sigma_1 \times \dots \times \sigma_k] \to \tau$, using abstraction, application and meta-variable application $Z[s_1,\dots,s_k]$ (where $Z : [\sigma_1 \times \dots \times \sigma_k] \to \tau \in \mathcal{M}$ and each $s_i : \sigma_i$). Meta-variables are not used as λ-binders. We denote $FMV(s)$ for the set of meta-variables occurring in s.

A *substitution* γ is a partial function mapping variables $x : \sigma$ to terms of type σ and meta-variables $Z : [\sigma_1 \times \dots \times \sigma_k] \to \tau$ to terms $\lambda x_1 \dots x_k.t$ of type $\sigma_1 \to \dots \to \sigma_k \to \tau$. For a meta-term s whose meta-variables all occur in the domain of γ, we let $s\gamma$ denote s with occurrences of variables x in the domain of γ replaced by $\gamma(x)$, and $Z[s_1,\dots,s_k]$ by $t[x_1 := s_1\gamma,\dots,x_k := s_k\gamma]$ if $\gamma(Z) = \lambda x_1 \dots x_k.t$; here, $[x_1 := s_1\gamma,\dots,x_k := s_k\gamma]$ is the substitution mapping each x_i to $s_i\gamma$.

Rules are pairs $\ell \Rightarrow r$ of meta-terms of the same type, such that $FMV(r) \subseteq FMV(\ell)$, both sides are *closed* (all their variable occurrences are bound) and ℓ is a *pattern*: for all sub-meta-terms of ℓ which have the form $Z[s_1,\dots,s_k]\ t_1 \cdots t_n$ necessarily $n = 0$ and $s_1,\dots,s_k$ are distinct variables. A set of rules $\mathcal{R}$ defines a *rewrite relation* $\Rightarrow_{\mathcal{R}}$ as the smallest monotonic relation on terms that contains all pairs $(\ell\gamma, r\gamma)$ with $\ell \Rightarrow r \in \mathcal{R}$ and γ a substitution on domain $FMV(\ell)$.

Meta-variables are used in early forms of higher-order rewriting such as Aczel's Contraction Schemes [1] and Klop's Combinatory Reduction Systems [16]. They strike a balance between matching modulo β-reduction and syntactic matching. Essentially, applying a substitution on a meta-term can be seen as computing a β-development. For example $\mathtt{d}\ (\lambda x.\mathtt{sin}\ (Z[x]))[Z := \lambda y.\mathtt{plus}\ y\ x]$ evaluates to $\mathtt{d}\ (\lambda z.\mathtt{sin}\ (\mathtt{plus}\ z\ x))$, and $Z[\mathtt{0},\mathtt{nil}][Z := \lambda xy.\mathtt{plus}\ x\ (\mathtt{len}\ y)]$ to $\mathtt{plus}\ \mathtt{0}\ (\mathtt{len}\ \mathtt{nil})$.

Example 1. 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:

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

Then $\mathtt{map}\ (\lambda x.\mathtt{0})\ (\mathtt{cons}\ \mathtt{0}\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{nil})) \Rightarrow_{\mathcal{R}} \mathtt{cons}\ \mathtt{0}\ (\mathtt{map}\ (\lambda x.\mathtt{0})\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{nil})) \Rightarrow_{\mathcal{R}} \mathtt{cons}\ \mathtt{0}\ (\mathtt{cons}\ \mathtt{0}\ (\mathtt{map}\ (\lambda x.\mathtt{0})\ \mathtt{nil})) \Rightarrow_{\mathcal{R}} \mathtt{cons}\ \mathtt{0}\ (\mathtt{cons}\ \mathtt{0}\ \mathtt{nil})$. Note that the bound variable x does not need to occur in the body of $\lambda x.\mathtt{0}$ to match $\lambda x.Z[x]$. It is *allowed* to occur, though: $\mathtt{map}\ (\lambda x.\mathtt{s}\ (\mathtt{s}\ x))\ (\mathtt{cons}\ \mathtt{0}\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{nil}))$ reduces in three steps to $\mathtt{cons}\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0}))\ (\mathtt{cons}\ (\mathtt{s}\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0})))\ \mathtt{nil})$.

Example 2. In Example 1, a term $\mathtt{map}\ \mathtt{s}\ (\mathtt{cons}\ \mathtt{0}\ \mathtt{nil})$ can *not* be reduced, because $\mathtt{s}$ does not instantiate $\lambda x.Z[x]$. We could alternatively consider the rules:

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

Here, Z has a type declaration $[] \to \mathtt{nat} \to \mathtt{nat}$ instead of $[\mathtt{nat}] \to \mathtt{nat}$, and the second rule employs explicit application. 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 may need explicit β-reductions; e.g., $\mathtt{map}\ (\lambda x.\mathtt{s}\ x)\ (\mathtt{cons}\ \mathtt{0}\ \mathtt{nil}) \Rightarrow_{\mathcal{R}} \mathtt{cons}\ ((\lambda x.\mathtt{s}\ x)\ \mathtt{0})\ (\mathtt{map}\ (\lambda x.\mathtt{s}\ x)\ \mathtt{nil}) \Rightarrow_{\beta} \mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ (\mathtt{map}\ (\lambda x.\mathtt{s}\ x)\ \mathtt{nil})$.

Following [19, §2.3.1] and [18, §7], *uncurrying* does not affect termination provided the rules are (essentially) unchanged. That is, let $arity(\mathtt{f})$ denote the largest number k such that (1) `f` can be applied to at least k arguments, and (2) every occurrence of `f` in $\mathcal{R}$ is applied to at least k arguments. Then to prove termination it suffices to show that there is no infinite reduction $s_1 \Rightarrow_{\mathcal{R}} s_2 \Rightarrow_{\mathcal{R}} \dots$ where, in every term s_i, all symbols `f` always occurs with at least $arity(\mathtt{f})$ arguments. In Example 1, $arity(s) = 1$ and $arity(\mathtt{cons}) = arity(\mathtt{map}) = 2$; thus, we do not need to consider terms such as `map s (cons 0 nil)` or `map` $(\lambda x.\mathtt{s}\ x)$ for termination. Arities are essential in various techniques (see, e.g., §4).

Input to WANDA.. WANDA is written for Linux, and can be invoked using:

```
    > ./wanda.exe <filename>
```

Filenames describing an AFSM should have extension `.afsm` and list first all function symbols with their types (each on an individual line) and then the rules, after an empty line as separator. Arities and types of meta-variables do not need to be given, as these are automatically derived. Types of bound variables *may* be given in the binder, but can typically also be derived from context.

Example 3. Example 1 can be described to WANDA as follows.

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

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

WANDA automatically derives arity 2 for both `cons` and `map`. Removing the `:nat` part in the rules does not affect the analysis, as this typing is clear from context.

The formalism used in the termination competition [27] – "higher-order rewriting union beta", which I will typically refer to as Algebraic Functional Systems (AFSs) – is very similar to AFSMs, but uses variables for matching rather than meta-variables; this gives rules like those in Example 2. WANDA can read such systems either in the competition's `.xml` format or in her own human-readable presentation of this same format (`.afs`), and translates them to AFSMs by applying the transformations of [18] and then replacing all free variables $x : \sigma$ in rules by a corresponding meta-variable $X : [] \to \sigma$.

The use of meta-variables for matching allows for the representation of rules such as $\mathtt{d}\ (\lambda x.\mathtt{sin}\ Z[x]) \Rightarrow \lambda x.(\mathtt{d}\ (\lambda y.Z[y])\ x) \times (\mathtt{cos}\ x)$, which have no counterpart in AFSs. This makes it possible to encode *pattern HRSs* [22, 24] and also *CRSs* [16] into AFSMs. It is future work to include such translations directly into WANDA.

Thus, the first step of a termination analysis in WANDA is to read the input: either an AFSM whose types are derived, or an AFS which is simplified and translated to the AFSM formalism. The second step is to derive arities, which are saved for later usage. When this is done, control is passed to the non-termination module.

3. Non-termination

Although the focus in her development has been on termination, WANDA has a basic checker with two tests to quickly identify simple cases of non-termination.

Obvious loops. An AFSM is clearly non-terminating if there is a reduction $s \Rightarrow_{\mathcal{R}}^* t$ such that an instance $s\gamma$ of s occurs as a subterm of t. To discover such loops, `WANDA` takes a rule left-hand side, replaces meta-variable applications $Z[x_1, \dots, x_k]$ by variable applications $y\ x_1 \cdots x_k$, and performs a breadth-first search on reducts, not going beyond the first thousand reducts. This simple method will not find any sophisticated counterexamples for termination, but is quick and easy, and often catches mistakes in a recursive call.

Instances of $\omega\omega$. Non-termination of the untyped λ-calculus may be demonstrated with the self-reducing term $\omega\omega$, where $\omega = \lambda x.xx$. A higher-order variation of this example is given by rules such as $\mathtt{f}\ (\mathtt{c}\ Z)\ X \Rightarrow Z\ X$ with $\mathtt{c} : \sigma \to \sigma$, where a function ($Z$) is taken out of a lower-order context: here, if we let $\omega := \mathtt{c}\ (\lambda x.\mathtt{f}\ x\ x)$, we have $\mathtt{f}\ \omega\ \omega \Rightarrow_{\mathcal{R}} (\lambda x.\mathtt{f}\ x\ x)\ \omega \Rightarrow_{\beta} \mathtt{f}\ \omega\ \omega$.

We can generalise the idea as follows. A *context* is a meta-term $\underline{C}[\square_1, \dots, \square_n]$ containing n typed holes $\square_i$, and $\underline{C}[s_1, \dots, s_n]$ denotes the same meta-term with each $\square_i$ replaced by s_i. `WANDA` identifies rules $\ell \Rightarrow r$ where ℓ has the form $\underline{C}[\underline{D}[Z], X]$ such that: (1) $Z : [] \to \sigma_1 \to \dots \to \sigma_n \to \tau \in \mathcal{M}$, with τ the type of ℓ; (2) there is i such that $\underline{D}[Z]$ has type σ_i and r can be written as $\underline{E}[Z\ s_1 \cdots X_i \cdots s_n]$ with $X_i : [] \to \sigma_i \in \mathcal{M}$; and (3) X and Z do not appear at other positions in $\underline{C}$ or $\underline{D}$. When such a rule is detected, `WANDA` uses it to construct a non-terminating term. She also looks for certain variations of this shape which consider meta-variables with more than 0 arguments.

Aside from these checks, the dependency pair framework employs a first-order termination tool to detect loops in the first-order rules of the AFSM as part of the DP framework (§5).

4. Rule removal

`WANDA`'s first step for proving termination is rule removal. The basic principle is simple: if we can identify a well-founded term ordering $\succ$ such that $s \succsim t$ whenever $s \Rightarrow_{\mathcal{R}} t$, and $s \succ t$ when a certain rule is used, then that rule cannot be an integral part of an infinite reduction, so can safely be removed – making the termination problem simpler. Rule removal is not *necessary* within `WANDA` (disabling it does not lose any benchmarks), but often leads to shorter runtimes and simpler proofs.

In practice, we do not use a single well-founded ordering but a *reduction pair*:

Definition 4. 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 $FMV(\mathtt{f}\ s_1 \cdots s_n) \cup FMV(t)$, then $s\gamma \succsim t\gamma$ (and similar for $\succ$);
- $\succsim$ is *monotonic*: if $s \succsim t$, then $s\ u \succsim t\ u$, $u\ s \succsim u\ t$ 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.

Reduction pairs also play a large rule in the dependency pair framework (§5); there, strong monotonicity is not required. However, depending on the query there may be additional requirements, such as $\mathtt{f}\ \vec{X} \succsim X_i$ for some of the symbols $\mathtt{f}$.

`WANDA` has two ways to generate reduction pairs: *weakly monotonic interpretations* and *recursive path orderings*. Both techniques extend first-order methods, and are most powerful when arity is taken into account. To do this in the most natural way, `WANDA` implicitly converts meta-terms which respect the *arity* function into *functional* notation, where applications are removed as follows:

$$\begin{array}{rcll}
\mathtt{uncurry}(x) & = & x & \text{if } x \text{ is a variable} \\
\mathtt{uncurry}(\lambda x.s) & = & \lambda x.\mathtt{uncurry}(s) & \\
\mathtt{uncurry}(Z[s_1,\dots,s_k]) & = & Z[\mathtt{uncurry}(s_1),\dots,\mathtt{uncurry}(s_k)] & \\
\mathtt{uncurry}(\mathtt{f}\ s_1\cdots s_k) & = & \mathtt{f}(\mathtt{uncurry}(s_1),\dots,\mathtt{uncurry}(s_k)) & \text{if } k = \mathit{arity}(\mathtt{f}) \\
\mathtt{uncurry}(s\ t) & = & @^{\langle\sigma,\tau\rangle}(\mathtt{uncurry}(s),\mathtt{uncurry}(t)) & \text{if } s : \sigma \to \tau \\
 & & \text{and } s \text{ does not have the form } \mathtt{f}\ s_1\cdots s_{\mathit{arity}(\mathtt{f})-1} &
\end{array}$$

Essentially, the rules are uncurried and applications replaced by explicit symbols. For $\mathtt{f} : \sigma_1 \to \dots \to \sigma_k \to \tau$, we consider $\mathtt{f}(s_1,\dots,s_k)$ as a (meta-)term of type τ.

Example 5. The uncurried version of the AFSM in Example 1 is:

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

The second rule in Example 2 is uncurried to:

$$\mathtt{map}(Z,\mathtt{cons}(H,T)) \Rightarrow \mathtt{cons}(@^{\langle\mathtt{nat},\mathtt{nat}\rangle}(Z,H),\mathtt{map}(Z,T))$$

4.1. Weakly monotonic algebras.

The idea of van de Pol's *weakly monotonic algebras* [26] 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 $[\![\mathtt{o}]\!]$ is the set of natural numbers for a base type $\mathtt{o}$ 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 follows.

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 $Z : [\sigma_1 \times \dots \times \sigma_n] \to \tau$ to an element of $[\![\sigma_1 \to \dots \to \sigma_n \to \tau]\!]$, we let $[s]^\mathcal{J}_\alpha$ be recursively defined as follows:

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

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 $\mathcal{J}_\mathtt{f}$ (including all $\mathcal{J}_{@^{\langle\sigma,\tau\rangle}}$) is monotonic over $>$ in the first $\mathit{arity}(\mathtt{f})$ arguments.

In [12], 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 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 $\mathcal{J}_{@_{\sigma,\tau}} = (f, n) \mapsto f(n) + n$. Then, writing $F := \alpha(Z)$, $n := \alpha(H)$, $m := \alpha(T)$:

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

Also, $\mathcal{J}_{@^{\langle\mathtt{nat},\mathtt{nat}\rangle}}(F, n) = F(n) + n \geq F(n)$, and we can similarly choose all $\mathcal{J}_{@^{\langle\sigma,\tau\rangle}}$ so that $\Rightarrow_\beta$ is included in $\succsim$. As all $\mathcal{J}_\mathtt{f}$ are strictly monotonic in all arguments, we may remove the second rule from Example 2.

4.2. `StarHorpo`. The recursive path ordering [8] is a syntactic method to extend an ordering on function symbols to an ordering on first-order terms. There are various extensions of RPO (e.g. [9, 15]) including a several higher-order variations (e.g. [5, 14]). `WANDA` uses her own definition, based on *iterative* path orders [17], which works well with meta-variables and (unlike older HORPOs) is natively transitive.

Following [17], `StarHorpo` employs a star mark $\star$ to indicate a decrease; intuitively, $\mathtt{f}^\star_\sigma(s_1, \dots, s_k)$ indicates an upper bound for all functional meta-terms of type σ which are *strictly smaller* than $\mathtt{f}(s_1, \dots, s_k)$. Let $s^\star$ denote $\lambda x_1 \dots x_n.\mathtt{f}^\star_\sigma(s_1, \dots, s_k)$ if $s = \lambda x_1 \dots x_n.\mathtt{f}(s_1, \dots, s_k)$. If s has any other form, then $s^\star$ is undefined.

`StarHorpo` assumes given a *precedence* $\underline{\blacktriangleright}$: a quasi-ordering on all symbols, whose strict part $\blacktriangleright$ is well-founded; we let $\approx$ denote the equivalence relation $\underline{\blacktriangleright} \cap \underline{\blacktriangleleft}$. We assume that there is a special symbol, $\bot$, which is minimal for $\underline{\blacktriangleright}$ (i.e., $\mathtt{f} \underline{\blacktriangleright} \bot$ for all $\mathtt{f}$). All symbols are assigned a *status* in $\{Lex, Mul\}$, and let $\succ^{\mathtt{f}}_\star$ denotes either the lexicographic or multiset extension of $\succ_\star$, depending on the status of $\mathtt{f}$. Then $(\succeq_\star, \succ_\star)$ is given by the following rules:

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

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 different numbers of arguments, but must always have at least *arity*($\mathtt{f}$). The notation $s\langle t_1, \dots, t_n\rangle$ indicates an "application": $s\langle\rangle = s$, $(\lambda x.s)\langle t, \vec{u}\rangle = s[x := t]\langle \vec{u}\rangle$ and $\mathtt{f}(\vec{s})\langle t, \vec{u}\rangle = \mathtt{f}^\star_{\sigma\to\tau}(\vec{s})\langle t, \vec{u}\rangle = \mathtt{f}^\star_\tau(\vec{s}, t)\langle \vec{u}\rangle$. Moreover, as part of `StarHorpo`, function symbols may have some of their arguments permutated or (if strong monotonicity is not required) filtered away; symbols with no remaining arguments may be mapped to $\bot_\sigma$ for a suitable σ.

The full explanation of these rules is available in [19, Chapter 5].

Example 7. To see that $\Rightarrow_\beta$ is included in $\succeq_\star$, note that $\succeq_\star$ is monotonic by (Fun), (FAbs) and (Meta), and we can derive: $@^{\langle\sigma,\tau\rangle}(\lambda x.Z[x], Y) \succeq_\star Z[Y]$ by (Put), because $@^{\langle\sigma,\tau\rangle\star}_\tau(\lambda x.Z[x], Y) \succeq_\star Z[Y]$ by (Select), because $Z[@^{\langle\sigma\to\tau\rangle\star}_\sigma(\lambda x.Z[x], Y)] \succeq_\star Z[Y]$ by (Meta), because $@^{\langle\sigma\to\tau\rangle\star}_\sigma(\lambda x.Z[x], Y) \succeq_\star Y$ by (Select), because $Y \succeq_\star Y$ by (Meta).

`WANDA` combines the search for a suitable precedence and status function with the search for a permutation and filtering, using a SAT encoding following [19, Chapter 8.6].

5. The higher-order dependency pair framework

If any rules remain after rule removal, `WANDA` passes them on to the *dependency pair framework*. Like the first-order DP framework [13], it is an extendable framework for termination and non-termination, which new termination methods can easily be plugged into in the form of processors. The DP framework is detailed in [10]. We here consider the high-level steps.

Delegation to a first-order prover. Following [11], the *first-order* rules in the AFSM are identified and passed to an external first-order termination tool. If this tool detects non-termination and returns a counterexample that can be typed (or if the AFSM is orthogonal, in which case the typing of the first-order part is irrelevant for its termination), `WANDA` concludes non-termination. If the first-order prover concludes termination, then all dependency pairs for these first-order rules are omitted for the remainder of the framework.

Static and Dynamic DPs. There are two approaches to generate dependency pairs, originating from distinct lines of work around the same period [20, 21]. In both cases, a set `DP` of "dependency pairs" (a kind of rewrite rules) is generated, and termination follows if there is no infinite chain $s_1, s_2, \dots$ with each $s_i \Rightarrow_{\mathtt{DP}} \cdot \Rightarrow_{\mathcal{R}}^* s_{i+1}$. Here, steps using $\Rightarrow_{\mathtt{DP}}$ may only be applied at the root of a term, and steps using $\Rightarrow_{\mathcal{R}}$ only in argument positions. A *DP problem* is the question whether a chain of a certain form exists, and *DP processors* simplify DP problems into easier ones – for example by removing DPs using a reduction pair.

The dynamic approach is always applicable, and could in theory be used also to prove non-termination – although in `WANDA` this is not yet done. The static approach is only applicable to AFSMs which pass certain restrictions, and may admit infinite chains even when the AFSM is terminating. However, proofs using the static approach are typically much simpler, since it does not generate "collapsing" DPs (of a form such as $\mathtt{f}\ \ell_1 \cdots \ell_n \Rightarrow Z[s_1]s_m$).

Despite their differences, the same processors apply to static and dynamic DPs;- the only difference is in their generation and whether the initial DP problem can be used to prove non-termination. Thus, `WANDA` uses the following strategy.

```
if not static_applicable($\mathcal{R}$) then return framework(static_DPs,$\mathcal{R}$);
else if static_DPs $\subseteq$ dynamic_DPs then return framework(static_DPs,$\mathcal{R}$);
else let tmp = framework(dynamic_DPs,$\mathcal{R}$) in
  if tmp = YES or tmp = NO then return tmp;
  else return framework(static_DPs,$\mathcal{R}$);
```

6. Conclusions and future work

Overall, `WANDA` takes an input file describing an AFSM (or an AFS), performs an analysis following Sections 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.

There are many directions for improvement. Most pertinently, due to the presence of a large database of termination benchmarks in the competition format [28], `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, and build on primarily higher-order techniques like sized types [3].

A complete discussion of most techniques in `WANDA` and the technology behind automating them is available in the author's PhD thesis [19]. `WANDA` is open-source and available from:

`http://wandahot.sourceforge.net/`

References

[1] P. Aczel. A general Church-Rosser theorem. Unpublished Manuscript, 1978.

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

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

[4] F. Blanqui, J. Jouannaud, and M. Okada. Inductive-data-type systems. *TCS*, 272(1-2):41–68, 2002.

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

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

[7] Community. The international Confluence Competition (CoCo). http://coco.nue.riec.tohoku.ac.jp/.

[8] N. Dershowitz. Orderings for term rewriting systems. *TCS*, 17(3):279–301, 1982.

[9] M. Ferreira and H. Zantema. Syntactical analysis of total termination. In *Proc. ALP*, volume 850 of *LNCS*, pages 204–222, 1994.

[10] C. Fuhs and C. Kop. The unified higher-order dependency pair framework. TODO.

[11] C. Fuhs and C. Kop. Harnessing first order termination provers using higher order dependency pairs. In *Proc. FroCoS*, volume 6989 of *LNAI*, pages 147–162, 2011.

[12] C. Fuhs and C. Kop. Polynomial interpretations for higher-order rewriting. In *Proc. RTA*, volume 15 of *LIPIcs*, pages 176–192, 2012.

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

[14] J. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proc. LICS*, IEEE, pages 402–411, 1999.

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

[16] J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *TCS*, 121(1-2):279 – 308, 1993.

[17] 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, 2006. Festschrift.

[18] C. Kop. Simplifying algebraic functional systems. In *Proc. CAI*, volume 6742 of *LNCS*, pages 201–215, 2011.

[19] C. Kop. *Higher Order Termination.* PhD thesis, VU University Amsterdam, 2012.

[20] C. Kop and F. van Raamsdonk. Dynamic dependency pairs for algebraic functional systems. *LMCS*, 8(2):10:1–10:51, 2012.

[21] K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE TIS*, 92(10):2007–2015, 2009.

[22] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *JLC*, 1(4):497–536, 1991.

[23] J. Nagele. CoCo 2016 participant: CSIˆho 0.2. http://coco.nue.riec.tohoku.ac.jp/2016/papers/csiho.pdf.

[24] T. Nipkow. Higher-order critical pairs. In *Proc. LICS*, pages 342–349, 1991.

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

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

[27] Wiki. Termination Portal. http://www.termination-portal.org/wiki/Termination_Competition.

[28] Wiki. Termination Problem DataBase (TPDB). http://termination-portal.org/wiki/TPDB.