Polynomial Interpretations for Higher-Order Rewriting [*]

Carsten Fuhs[1] and Cynthia Kop[2]

1 University College London, Gower Street, London WC1E 6BT, UK
2 Vrije Universiteit, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

Abstract

The termination method of weakly monotonic algebras, which has been defined for higher-order rewriting in the HRS formalism, offers a lot of power, but has seen little use in recent years. We adapt and extend this method to the alternative formalism of *algebraic functional systems*, where the simply-typed λ-calculus is combined with algebraic reduction. Using this theory, we define *higher-order polynomial interpretations*, and show how the implementation challenges of this technique can be tackled. A full implementation is provided in the termination tool WANDA.

Keywords and phrases higher-order rewriting, termination, polynomial interpretations, weakly monotonic algebras, automation

1 Introduction

One of the most prominent techniques in termination proofs for first-order term rewriting systems (TRSs) is the use of *polynomial interpretations*. In this method, which dates back to the seventies [24], terms are mapped to polynomials over (e.g.) $\mathbb{N}$. The method is quite intuitive, since a TRS is usually written with a meaning for the function symbols in mind, which can often be modeled by the interpretation. In addition, it has been implemented in various automatic tools, such as AProVE [14], $\mathsf{T_TT_2}$ [22] and Jambox [8]. Polynomial interpretations are an instance of the *monotonic algebra* approach [9] which also includes for instance *matrix interpretations*. They are used both on their own, and in combination with *dependency pair* approaches [1].

In the higher-order world, monotonic algebras were among the first termination methods to be defined, appearing as early as 1994 [26]; an in-depth study is done in van de Pol's 1996 PhD thesis [27]. Surprisingly, the method has been almost entirely absent from the literature ever since. This is despite a lot of interest in higher-order rewriting, witnessed not only by a fair number of publications, but also by the recent participation of higher-order tools in the *annual Termination Competition* [30]. Since the addition of a higher-order category, two tools have participated: THOR [4], by Borralleras and Rubio, and WANDA [18], by the second author of this paper. So far, neither tool has implemented weakly monotonic algebras.

In this paper we aim to counteract this situation, by both studying the class of polynomial interpretations in the natural numbers, and implementing the resulting technique in the termination tool WANDA. Van de Pol did not consider automation of his method (there was less focus on automation at the time), but there are now years of experience of the first-order world to build on; we will lift the parametric first-order approach [6], and make some necessary adaptations to cater for the presence of higher-order variables.

* This research is supported by the Netherlands Organisation for Scientific Research (NWO-EW) under grant 612.000.629 (HOT).

Paper Setup Section 2 discusses preliminaries: *Algebraic Functional Systems*, the higher-order formalism we consider, reduction pairs and weakly monotonic algebras for typed λ-terms. In Section 3 we extend these definitions to AFSs, and define a general termination method. Section 4 defines the class of *higher-order polynomials*, and in Section 5 we show how suitable polynomial interpretations can be found automatically. Experiments with this implementation are presented in Section 6, and an overview and ideas for future work are given in Section 7.

The main contribution of this paper are the techniques for automation, discussed in Section 6. For simplicity of the code, these techniques are limited to the (very common) class of second-order AFSs, although extensions to systems of a higher order are possible. As far as we know, this is the first implementation of higher-order polynomial interpretations.

An extended version of this paper with more elaborate proofs is available at [13].

2 Background

2.1 Algebraic Functional Systems

We consider algebraic higher-order rewriting as defined by Jouannaud and Okada, also called *Algebraic Functional Systems (AFSs)* [16]. This formalism combines the simply-typed λ-calculus with algebraic reduction, and appears in papers on e.g. HORPO [17], MHOSPO [5] and dependency pairs [20]; it is also the formalism in the higher-order category of the annual termination competition. We follow roughly the definitions in [29, Ch. 11.2.3], as recalled below.

Types and Terms The set of *simple types* (or just *types*) is generated from a given set $\mathcal{B}$ of *base types* and the binary, right-associative type constructor $\Rightarrow$; types are denoted by σ, τ and base types by ι, κ. A type with at least one occurrence of $\Rightarrow$ is called a *functional type*. A *type declaration* is an expression of the form $[\sigma_1 \times \ldots \times \sigma_n] \Rightarrow \tau$ for types σ_i, τ; if $n = 0$ we just write τ. Type declarations are not types, but are used to "type" function symbols. All types can be expressed in the form $\sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow \iota$ (with $n \geq 0$ and $\iota \in \mathcal{B}$). The *order* of a type is $order(\iota) = 0$ if $\iota \in \mathcal{B}$, and $order(\sigma \Rightarrow \tau) = \max(order(\sigma)+1, order(\tau))$. Extending this to type declarations, $order([\sigma_1 \times \ldots \times \sigma_n] \Rightarrow \tau) = \max(order(\sigma_1) + 1, \ldots, order(\sigma_n) + 1, order(\tau))$.

We assume a set $\mathcal{V}$ of infinitely many typed variables for each type, and a set $\mathcal{F}$ disjoint from $\mathcal{V}$ which consists of function symbols, each equipped with a type declaration. *Terms* over $\mathcal{F}$ are those expressions s for which we can infer $s : \sigma$ for some type σ using the clauses:

`(var)`	$x : \sigma$	if $x : \sigma \in \mathcal{V}$
`(app)`	$s \cdot t : \tau$	if $s : \sigma \Rightarrow \tau$ and $t : \sigma$
`(abs)`	$\lambda x. s : \sigma \Rightarrow \tau$	if $x : \sigma \in \mathcal{V}$ and $s : \tau$
`(fun)`	$f(s_1, \ldots, s_n) : \tau$	if $f : [\sigma_1 \times \ldots \times \sigma_n] \Rightarrow \tau \in \mathcal{F}$ and $s_1 : \sigma_1, \ldots, s_n : \sigma_n$

Note that a function symbol $f : [\sigma_1 \times \ldots \times \sigma_n] \Rightarrow \tau$ takes exactly n arguments, and τ is not necessarily a base type (a type declaration gives the *arity* of the symbol). λ binds occurrences of variables as in the λ-calculus. Terms are considered modulo α-conversion; bound variables are renamed if necessary. Variables which are not bound are called *free*, and the set of free variables of s is denoted $FV(s)$. Application is left-associative, so $s \cdot t \cdot u$ should be read $(s \cdot t) \cdot u$. Terms constructed without clause `(fun)` are also called *(simply-typed)* λ*-terms*.

A *substitution* $[\vec{x} := \vec{s}]$, with $\vec{x}$ and $\vec{s}$ finite vectors of equal length, is the homomorphic extension of the type-preserving mapping $\vec{x} \mapsto \vec{s}$ from variables to terms. Substitutions are denoted γ, δ, and the result of applying γ to a term s is denoted $s\gamma$. The *domain* $\mathsf{dom}(\gamma)$ of $\gamma = [\vec{x} := \vec{s}]$ is $\{\vec{x}\}$. Substituting does not bind free variables. A *context* $C[]$ is a term with a single occurrence of a special symbol $\square_\sigma$. The result of replacing $\square_\sigma$ in $C[]$ by a term s of type σ is denoted $C[s]$. Free variables may be captured; if $C[] = \lambda x. \square_\sigma$ then $C[x] = \lambda x. x$.

Rules and Rewriting A *rewrite rule* is a pair of terms $l \to r$ such that l and r have the same type and all free variables of r also occur in l. In [19] some termination-preserving transformations on the general format of AFS-rules are presented; using these results, we may additionally assume that l has the form $f(l_1, \dots, l_n) \cdot l_{n+1} \cdots l_m$ (with $f \in \mathcal{F}$ and $m \geq n \geq 0$), that l has no subterms $x \cdot s$ with x a free variable, and that neither l nor r have a subterm $(\lambda x.\, s) \cdot t$. Given a set of rules $\mathcal{R}$, the *rewrite* or *reduction relation* $\to_{\mathcal{R}}$ on terms is given by the following clauses:

`(rule)`	$C[l\gamma]$	$\to_{\mathcal{R}}$	$C[r\gamma]$	with $l \to r \in \mathcal{R}$, C a context, γ a substitution
(β)	$C[(\lambda x.\, s) \cdot t]$	$\to_{\mathcal{R}}$	$C[s[x := t]]$	with s, t terms, C a context

An *algebraic functional system (AFS)* is the combination of a set of terms and a rewrite relation on this set, and is usually specified by a pair $(\mathcal{F}, \mathcal{R})$, or just by a set $\mathcal{R}$ of rules. An AFS is terminating if there is no infinite reduction $s_1 \to_{\mathcal{R}} s_2 \to_{\mathcal{R}} \dots$

An AFS is *second-order* if the type declarations of all function symbols have order ≤ 2. In a second-order system, all free variables in the rules have order ≤ 1 (this follows by the restrictions on the left-hand side), and all bound variables have base type (this holds because free variables have order ≤ 1 and we have assumed that the rules do not contain β-redexes).

▶ **Example 2.1.** One of the examples considered in this paper is the AFS $\mathsf{shuffle}$. This (second-order) system for list manipulation has five function symbols, $\mathsf{nil} : \mathsf{natlist}$, $\mathsf{cons} : [\mathsf{nat} \times \mathsf{natlist}] \Rightarrow \mathsf{natlist}$, $\mathsf{append} : [\mathsf{natlist} \times \mathsf{natlist}] \Rightarrow \mathsf{natlist}$, $\mathsf{reverse} : [\mathsf{natlist}] \Rightarrow \mathsf{natlist}$, $\mathsf{shuffle} : [(\mathsf{nat} \Rightarrow \mathsf{nat}) \times \mathsf{natlist}] \Rightarrow \mathsf{natlist}$, and the following rules:

$$\begin{array}{rclrcl}
\mathsf{append}(\mathsf{cons}(h,t),l) & \to & \mathsf{cons}(h,\mathsf{append}(t,l)) & \mathsf{append}(\mathsf{nil},l) & \to & l \\
\mathsf{reverse}(\mathsf{cons}(h,t)) & \to & \mathsf{append}(\mathsf{reverse}(t),\mathsf{cons}(h,\mathsf{nil})) & \mathsf{reverse}(\mathsf{nil}) & \to & \mathsf{nil} \\
\mathsf{shuffle}(F,\mathsf{cons}(h,t)) & \to & \mathsf{cons}(F \cdot h,\mathsf{shuffle}(F,\mathsf{reverse}(t))) & \mathsf{shuffle}(F,\mathsf{nil}) & \to & \mathsf{nil}
\end{array}$$

2.2 Reduction Pairs

To prove termination, modern approaches typically use *reduction pairs*, in one of three setups:

For *rule removal*, we consider a *strong reduction pair*: a pair $(\succsim, \succ)$ of a quasi-ordering and a well-founded ordering on terms, such that $\succsim$ and $\succ$ are *compatible*: $\succsim \cdot \succ$ is included in $\succ$ or $\succ \cdot \succsim$ is, both $\succsim$ and $\succ$ are *monotonic*, both $\succsim$ and $\succ$ are *stable* (preserved under substitution), and in the higher-order case, $\succsim$ *contains* β: $(\lambda x.\, s) \cdot t \succsim s[x := t]$.

If $\mathcal{R} = \mathcal{R}_1 \uplus \mathcal{R}_2$ and $l \succ r$ for rules in $\mathcal{R}_1$, and $l \succsim r$ for rules in $\mathcal{R}_2$, then there is no $\to_{\mathcal{R}}$-sequence which uses the rules in $\mathcal{R}_1$ infinitely often; this would contradict well-foundedness of $\succ$. Thus, $\to_{\mathcal{R}}$ is terminating if $\to_{\mathcal{R}_2}$ is terminating. In practice, we try to orient all rules with either $\succ$ or $\succsim$, and then remove those ordered with $\succ$ and continue with the rest.

The second setup, *dependency pairs*, is more sophisticated. In this approach, dependency pair chains are considered, which use infinitely many "dependency pair" steps at the top of a term. It is enough to orient the resulting constraints with a *weak reduction pair*: a pair $(\succsim, \succ)$ of a quasi-ordering and a compatible well-founded ordering where both are stable, and $\succsim$ is monotonic and contains β. The dependency pair approach was defined for first-order TRSs in [1], and has seen many extensions and improvements since. For higher-order rewriting, two variations exist: *static dependency pairs* [23] and *dynamic dependency pairs* [28, 21].

The static dependency pair approach is restricted to *plain function passing* systems; slightly simplified, whenever a higher-order variable F occurs in the right-hand side of a rule $f(l_1, \dots, l_n) \to r$, then F is one of the l_i. Static dependency pairs may have variables in the right-hand side which do not occur in the left (such as a dependency pair $\mathsf{I}^\sharp(\mathsf{s}(n)) \to \mathsf{I}^\sharp(m)$), but always have the form $f^\sharp(l_1, \dots, l_n) \to g^\sharp(r_1, \dots, r_m)$. The static approach gives constraints of the form $l \succsim r$ or $l \succ r$ for dependency pairs $l \to r$, and $l \succsim r$ for rules $l \to r$.

The dynamic dependency pair approach is unrestricted, but right-hand sides of dependency pairs may be headed by a variable, e.g. $\mathsf{collapse}^\sharp(\mathsf{cons}(F, t)) \to F \cdot \mathsf{collapse}(t)$, and sometimes subterm steps are needed. Thus, the dynamic approach not only gives constraints $l \succ r$ or $l \succsim r$ for dependency pairs and $l \succsim r$ for rules, but also two further groups of constraints:

- $f(s_1, \ldots, s_n) \cdot t_1 \cdots t_m \succsim s_i \cdot \mathsf{c}_{\sigma_1} \cdots \mathsf{c}_{\sigma_{k_i}}$ if both sides have base type, $s_i : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_{k_i} \Rightarrow \iota$ and f is a symbol in some fixed set S (the c_{σ_j} are special symbols which may occur in the right-hand sides of dependency pairs but do not occur in the rules)
- $s \cdot t_1 \cdots t_n \succsim t_i \cdot \mathsf{c}_{\sigma_1} \cdots \mathsf{c}_{\sigma_{k_i}}$ if both sides have base type, and $t_i : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_{k_i} \Rightarrow \iota$

A third setup, which also uses a sort of reduction pair rather than the traditional reduction ordering, are the *monotonic semantic path orderings* from Borralleras and Rubio [5]. This method is based on a recursive path ordering, but uses a well-founded order on terms rather than a precedence on function symbols; this gives constraints of the form $s \succeq_I t$, $s \succeq_Q t$, $s \succ_Q t$, where $\succeq_I$ and $\succeq_Q$ are quasi-orderings and $s \succeq_I t$ implies $f(\ldots, s, \ldots) \succeq_Q f(\ldots, t, \ldots)$.

In this paper, we focus on the first two setups, which have been implemented in WANDA. However, the technique could be used with the monotonic semantic path ordering as well.

2.3 First-order Monotonic Algebras - Idea Sketch

In the first-order definition of monotonic algebras [9], terms are mapped to elements of a well-founded target domain $(A, >, \geq)$. This is done by choosing an interpretation function $\mathcal{J}(f)$ for all function symbols f that is monotonic w.r.t. $>$ and $\geq$, and extending this homomorphically to an interpretation $[\![\cdot]\!]$ of terms; for *polynomial interpretations*, $\mathcal{J}(f)$ is always a polynomial. If $[\![l]\!]_{\mathcal{J},\alpha} > [\![r]\!]_{\mathcal{J},\alpha}$ for all valuations α of the free variables of l, then $[\![C[l\gamma]]\!]_{\mathcal{J}} > [\![C[r\gamma]]\!]_{\mathcal{J}}$ for all contexts C and substitutions γ. Thus, the pair $(\succsim, \succ)$ where $s \succsim t$ if $[\![s]\!]_{\mathcal{J},\alpha} \geq [\![t]\!]_{\mathcal{J},\alpha}$ and $s \succ t$ if $[\![s]\!]_{\mathcal{J},\alpha} > [\![t]\!]_{\mathcal{J},\alpha}$ can be used as a strong reduction pair.

For example, to prove termination of the TRS consisting of the two append rules from Example 2.1, we might assign the following interpretation to the function symbols: $\mathcal{J}(\mathsf{nil}) = 2$, $\mathcal{J}(\mathsf{cons}) = \boldsymbol{\lambda} nm.n + m + 1$ and $\mathcal{J}(\mathsf{append}) = \boldsymbol{\lambda} nm.2 \cdot n + m + 1$. Here, the $\boldsymbol{\lambda}$ syntax indicates function creation: cons, for instance, is mapped to a function which takes two arguments, and returns their sum plus one. Calculating all $[\![l]\!]_{\mathcal{J},\alpha}, [\![r]\!]_{\mathcal{J},\alpha}$, and noting that $(\mathbb{N}, >, \geq)$ is a well-founded set and that all interpretations are monotonic functions, we see that the TRS is terminating because for all h, t, l: $4{+}l{+}1 > l$ (for the rule $\mathsf{append}(\mathsf{nil}, l) \to l$), and $2{\cdot}h{+}2{\cdot}t{+}2{+}l{+}1 > h{+}2{\cdot}t{+}l{+}1$ (for $\mathsf{append}(\mathsf{cons}(h, t), l) \to \mathsf{cons}(h, \mathsf{append}(t, l))$).

2.4 Weakly Monotonic Functionals

In higher-order rewriting we have to deal with infinitely many types (due to the type constructor $\Rightarrow$), a complication not present in first-order rewriting. As a consequence, it is not practical to map all terms to the same target set. A more natural interpretation would be, for instance, to map a functional term $\lambda x. s : \mathsf{o} \Rightarrow \mathsf{o}$ to an element of the function space $\mathbb{N} \Rightarrow \mathbb{N}$. However, this choice has problems of its own, since it forces the termination prover to deal with functions that absolutely nothing is known about. Instead, the target domain for interpreting terms, as proposed by van de Pol in [27], is the class of *weakly monotonic functionals*. To each type σ we assign a set $\mathcal{WM}_\sigma$ and two relations: a well-founded ordering $\sqsupset_\sigma$ and a quasi-ordering $\sqsupseteq_\sigma$. Intuitively, the elements of $\mathcal{WM}_{\sigma \Rightarrow \tau}$ are functions which preserve $\sqsupseteq$.

▶ **Definition 2.2** (Weakly Monotonic Functionals). [27, Def. 4.1.1] We assume given a *well-founded set*: a triple $\mathcal{A} = (A, >, \geq)$ of a non-empty set, a well-founded partial ordering on that set and a compatible quasi-ordering.[1] To each type σ we associate a set $\mathcal{WM}_\sigma$ of *weakly monotonic functionals of type* σ and two relations $\sqsupset_\sigma$ and $\sqsupseteq_\sigma$, defined inductively as follows:

For a base type ι, we have $\mathcal{WM}_\iota = A$; $\sqsupset_\iota = >$, and $\sqsupseteq_\iota = \geq$.

For a functional type $\sigma \Rightarrow \tau$, $\mathcal{WM}_{\sigma\Rightarrow\tau}$ consists of the functions f from $\mathcal{WM}_\sigma$ to $\mathcal{WM}_\tau$ such that: if $x \sqsupseteq_\sigma y$ then $f(x) \sqsupseteq_\tau f(y)$. Let $f \sqsupset_{\sigma\Rightarrow\tau} g$ iff $f(x) \sqsupset_\tau g(x)$, and $f \sqsupseteq_{\sigma\Rightarrow\tau} g$ iff $f(x) \sqsupseteq_\tau g(x)$ for all $x, y \in \mathcal{WM}_\sigma$.

Thus, $\mathcal{WM}_{\sigma\Rightarrow\tau}$ is a subset of the function space $\mathcal{WM}_\sigma \Rightarrow \mathcal{WM}_\tau$, consisting of functions which preserve $\sqsupseteq$. Note that both $\mathcal{WM}_\sigma$ and the relations $\sqsupset_\sigma$ and $\sqsupseteq_\sigma$ should be considered as parametrised with $\mathcal{A}$; the complete notation would be $(\mathcal{WM}^{\mathcal{A}}_\sigma, \sqsupset^{\mathcal{A}}_\sigma, \sqsupseteq^{\mathcal{A}}_\sigma)$. For readability, $\mathcal{A}$ will normally be omitted, as will the type denotations for the various $\sqsupset_\sigma$ and $\sqsupseteq_\sigma$ relations. The phrase "f is weakly monotonic" means that $f \in \mathcal{WM}_\sigma$ for some σ.

It is not hard to see that an element $\boldsymbol{\lambda} x_1 \dots x_n.P(x_1,\dots,x_n)$ of the function space $\mathcal{WM}_{\sigma_1} \Rightarrow \dots \Rightarrow \mathcal{WM}_{\sigma_n} \Rightarrow A$ is weakly monotonic if and only if:

$$\begin{array}{c}\forall N_1, M_1 \in \mathcal{WM}_{\sigma_1}, \dots, N_n, M_n \in \mathcal{WM}_{\sigma_n} : \\ \text{if each } N_i \sqsupseteq M_i \text{ then } P(N_1,\dots,N_n) \sqsupseteq P(M_1,\dots,M_n)\end{array}$$

By Lemmas 4.1.3 and 4.1.4 in [27] we obtain several pleasant properties of $\sqsupseteq$ and $\sqsupset$:

▶ **Lemma 2.3.** *For all types σ, the relations $\sqsupset_\sigma$ and $\sqsupseteq_\sigma$ are compatible, $\sqsupset_\sigma$ is well founded, $\sqsupseteq_\sigma$ is reflexive, and both $\sqsupset_\sigma$ and $\sqsupseteq_\sigma$ are transitive.*

Comment: the definition in [27] actually assigns a different set $\mathcal{A}_\iota$ to each base type ι (although there must be an addition operator $+_{\iota,\kappa,\iota}$ for every pair of base types). We use the same set for all base types, as this gives a simpler definition, and it is not obvious that using different sets gives a stronger technique; we could for instance choose $\mathcal{A} = \mathcal{A}_\iota \uplus \mathcal{A}_\kappa$ instead.

Also, in [27] $\mathcal{WM}_{\sigma\Rightarrow\tau}$ consists of functions f in a larger function space $\mathcal{I}_\sigma \Rightarrow \mathcal{I}_\tau$[2] such that $f(x) \in \mathcal{WM}_\tau$ if $x \in \mathcal{WM}_\sigma$ and f preserves $\sqsupseteq$. Our definition is essentially equivalent; every function in $\mathcal{WM}_\sigma \Rightarrow \mathcal{WM}_\tau$ can be extended to a function in $\mathcal{I}_\sigma \Rightarrow \mathcal{I}_\tau$.

▶ **Example 2.4** (Some Examples of Weakly Monotonic Functionals).

1. *Constant Function:* For all $n \in A$ and types $\tau = \tau_1 \Rightarrow \dots \Rightarrow \tau_k \Rightarrow \iota$, let $n_\tau := \boldsymbol{\lambda}\vec{x}.n$. Then $n_\tau \in \mathcal{WM}_\tau$, since $n_\tau(N_1,\dots,N_k) = n \sqsupseteq n = n_\tau(M_1,\dots,M_k)$ if all $N_i \sqsupseteq M_i$.
2. *Lowest Value Function:* Suppose A has a minimal element 0 for the ordering $>$. Then for any type $\tau = \tau_1 \Rightarrow \dots \Rightarrow \tau_k \Rightarrow \iota$ the function $\boldsymbol{\lambda} f.f(\vec{0})$, which maps $f \in \mathcal{WM}_\tau$ to $f(0_{\tau_1},\dots,0_{\tau_k})$ (where each 0_{τ_i} is a constant function), is in $\mathcal{WM}_{\tau\Rightarrow o}$ by induction on k.
3. *Maximum Function:* In the natural numbers, the function max which assigns to any two numbers the highest of the two is weakly monotonic, since $\max(a,b) \geq \max(a',b')$ if $a \geq a'$ and $b \geq b'$. For any type $\tau = \tau_1 \Rightarrow \dots \Rightarrow \tau_k \Rightarrow \iota$ (with $\iota \in \mathcal{B}$) let $\max_\tau(f,m) = \boldsymbol{\lambda} x_1 \dots x_k. \max(f(x_1,\dots,x_k), m)$. This function is in $\mathcal{WM}_{\tau\Rightarrow\iota\Rightarrow\tau}$ by induction on k.

The constant and lowest value function appear in [27]; the maximum function appears in [20].

▶ **Definition 2.5** (Interpreting a λ-Term to a Weakly Monotonic Functional)**.** Given a well-founded set $\mathcal{A} = (A, >, \geq)$, a simply-typed λ-term s and a *valuation* α which assigns to all variables $x : \sigma$ in $FV(s)$ an element of $\mathcal{WM}_\sigma$, let $[s]_\alpha$ be defined by the following clauses:

$$\begin{array}{lcll}[x]_\alpha & = & \alpha(x) & \text{if } x \in \mathcal{V} \\ [s \cdot t]_\alpha & = & [s]_\alpha([t]_\alpha) & \\ [\lambda x.\, s]_\alpha & = & \boldsymbol{\lambda} n.[s]_{\alpha\cup\{x\mapsto n\}} & \text{if } x \notin \mathsf{dom}(\alpha) \ \ \text{(always applicable with } \alpha\text{-conversion)}\end{array}$$

[1] Van de Pol defines $\geq$ as the reflexive closure of $>$. In contrast, here we generalise the notion of a well-founded set to include an explicitly given compatible quasi-ordering $\geq$.

[2] Here, $\mathcal{I}_\iota = \mathcal{A}_\iota$ if $\iota \in \mathcal{B}$, and $\mathcal{I}_{\sigma\Rightarrow\tau}$ is the full function space $\mathcal{I}_\sigma \Rightarrow \mathcal{I}_\tau$.

Definition 2.5 is an instance of a definition in [27] which suffices for the extension to AFSs. By Lemma 3.2.1 and Proposition 4.1.5(1) in [27], we have:

▶ **Lemma 2.6** (Facts on λ-Term Interpretations).

1. (Substitution Lemma) *Given a substitution* $\gamma = [x_1 := s_1, \dots, x_n := s_n]$ *and a valuation* α *whose domain does not include the* x_i*:* $[s\gamma]_\alpha = [s]_{\alpha \circ \gamma}$*. Here,* $\alpha \circ \gamma$ *is the valuation* $\alpha \cup \{x_1 \mapsto [s_1]_\alpha, \dots, x_n \mapsto [s_n]_\alpha\}$.
2. *If* $s : \sigma$ *is a simply-typed* λ*-term, then* $[s]_\alpha \in \mathcal{WM}_\sigma$ *for all valuations* α.

3 (Weakly and Extended) Monotonic Algebras for AFSs

The theory in [27] was defined for Nipkow's formalism of *Higher-order Rewrite Systems (HRSs)* [25], which differs in several ways from our *Algebraic Functional Systems.* Most importantly, in the setting of HRSs terms are equivalence classes modulo β; thus, the definitions in [27] are designed so that $[\![s]\!] = [\![t]\!]$ if s and t are equal modulo β. This is not convenient for AFSs, since then for instance $[\![(\lambda x.\, 0) \cdot t]\!] = [\![(\lambda x.\, 0) \cdot u]\!]$ regardless of t and u.

Fortunately, we do not need to redesign the whole theory for use with AFSs; rather, we can transpose the result using a transformation. We will need no more than Lemma 2.6.

Note: some of the results of this section have also been stated in [20], but the results there are limited to what is needed for the dynamic dependency pair approach; here, we are more general, by not fixing the interpretation of application and also studying strong monotonicity.

▶ **Definition 3.1** (Weakly Monotonic Algebras for AFSs). A *weakly monotonic algebra* for an AFS with function symbols $\mathcal{F}$ consists of a well-founded set $\mathcal{A} = (A, >, \geq)$ and an *interpretation function* $\mathcal{J}$ which assigns an element of $\mathcal{WM}_{\sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \tau}$ to all $f : [\sigma_1 \times \dots \times \sigma_n] \Rightarrow \tau \in \mathcal{F}$, and a value in $\mathcal{WM}_{\sigma \Rightarrow \sigma}$ to the fresh symbol $@^\sigma$ for all functional types σ.

Given an algebra $(\mathcal{A}, \mathcal{J})$, a term s over $\mathcal{F}$ and a *valuation* α which assigns to all variables $x : \sigma$ in $FV(s)$ an element of $\mathcal{WM}_\sigma$, let $[\![s]\!]_{\mathcal{J},\alpha}$ be defined recursively as follows:

$$\begin{array}{lcll}
[\![x]\!]_{\mathcal{J},\alpha} & = & \alpha(x) & \text{if } x \in \mathcal{V} \\
[\![f(s_1, \dots, s_n)]\!]_{\mathcal{J},\alpha} & = & \mathcal{J}(f)([\![s_1]\!]_{\mathcal{J},\alpha}, \dots, [\![s_n]\!]_{\mathcal{J},\alpha}) & \text{if } f \in \mathcal{F} \\
[\![s \cdot t]\!]_{\mathcal{J},\alpha} & = & \mathcal{J}(@^\sigma)([\![s]\!]_{\mathcal{J},\alpha}, [\![t]\!]_{\mathcal{J},\alpha}) & \text{if } s : \sigma \\
[\![\lambda x.\, s]\!]_{\mathcal{J},\alpha} & = & \boldsymbol{\lambda} n.[\![s]\!]_{\mathcal{J},\alpha \cup \{x \mapsto n\}} & \text{if } x \notin \mathsf{dom}(\alpha)
\end{array}$$

This definition, which roughly follows the ideas of [27] and extends the definition of a weakly monotone algebra in [9] to the setting of AFSs, assigns to every function symbol and variable a weakly monotonic functional, and calculates the value of the term accordingly. For the purposes of the interpretation, application is treated as a function symbol $@^\sigma$. As in [27], the interpretation function $\mathcal{J}$ is separate from the valuation α, as we will quantify over α.

▶ **Example 3.2.** Consider the shuffle signature from Example 2.1, extended with symbols 0 and s for the natural numbers. Let $\mathcal{A} = (\mathbb{N}, >, \geq)$. By way of example, choose: $\mathcal{J}(0) = 1$, $\mathcal{J}(\mathsf{s}) = \boldsymbol{\lambda} n.n + 2$, $\mathcal{J}(\mathsf{cons}) = \boldsymbol{\lambda} nm.n + m$, $\mathcal{J}(\mathsf{shuffle}) = \boldsymbol{\lambda} Fn.F(n)$ and $\alpha(z) = 37$. Then $[\![\mathsf{shuffle}(\lambda x.\, \mathsf{s}(x), \mathsf{cons}(\mathsf{s}(0), z))]\!]_{\mathcal{J},\alpha} = [\![F(n)]\!]_{\mathcal{J},\{F \mapsto \boldsymbol{\lambda} m.m+2, n \mapsto 40\}} = 42$.

▶ **Lemma 3.3** (Weakly Monotonic Algebras for AFSs). *Let* $(\mathcal{A}, \mathcal{J})$ *be a weakly monotonic algebra for* $\mathcal{F}$*, and* s, t *terms over* $\mathcal{F}$*. For all valuations* α *as described in Definition 3.1:*

1. $[\![s]\!]_{\mathcal{J},\alpha} \in \mathcal{WM}_\sigma$ *if* $s : \sigma$.
2. $[\![s]\!]_{\mathcal{J},\alpha \circ \gamma} = [\![s\gamma]\!]_{\mathcal{J},\alpha}$ *(where* $\alpha \circ \gamma = \alpha \cup \{x \mapsto [\![\gamma(x)]\!]_{\mathcal{J},\alpha} \mid x \in \mathsf{dom}(\gamma)\}$*)*
3. *If* $[\![s]\!]_{\mathcal{J},\delta} \sqsupseteq [\![t]\!]_{\mathcal{J},\delta}$ *for all valuations* δ*, then* $[\![s\gamma]\!]_{\mathcal{J},\alpha} \sqsupseteq [\![t\gamma]\!]_{\mathcal{J},\alpha}$.
   If $[\![s]\!]_{\mathcal{J},\delta} \sqsupset [\![t]\!]_{\mathcal{J},\delta}$ *for all valuations* δ*, then* $[\![s\gamma]\!]_{\mathcal{J},\alpha} \sqsupset [\![t\gamma]\!]_{\mathcal{J},\alpha}$.
4. *If* $[\![s]\!]_{\mathcal{J},\delta} \sqsupseteq [\![t]\!]_{\mathcal{J},\delta}$ *for all valuations* δ*, then* $[\![C[s]]\!]_{\mathcal{J},\alpha} \sqsupseteq [\![C[t]]\!]_{\mathcal{J},\alpha}$.

Proof. The proof proceeds by translating (arbitrary) terms to simply-typed λ-terms, and then reusing the original result. Interpretation of function symbols ($\mathcal{J}$) is translated to assignment of variables (α), and application is treated as a function symbol.

Consider the following transformation:

$$\begin{array}{rclrcll} \varphi(x) & = & x \ \ (x \in \mathcal{V}) & \varphi(f(s_1,\dots,s_n)) & = & x_f \cdot \varphi(s_1) \cdots \varphi(s_n) & (f \in \mathcal{F}) \\ \varphi(\lambda x.\, s) & = & \lambda x.\, \varphi(s) & \varphi(s \cdot t) & = & x_{@,\sigma} \cdot \varphi(s) \cdot \varphi(t) & (s : \sigma) \end{array}$$

Here, the x_f is a new variable of type $\sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \tau$ for $f : [\sigma_1 \times \dots \times \sigma_n] \Rightarrow \tau \in \mathcal{F}$, and $x_{@,\sigma}$ is a variable of type $\sigma \Rightarrow \sigma$. For any substitution γ, let γ^φ denote the substitution $[x := \varphi(\gamma(x)) \mid x \in \mathsf{dom}(\gamma)]$ (the x_f are left alone). We make the following observations:

(**) $\varphi(s\gamma) = \varphi(s)\gamma^\varphi$ *for all substitutions* γ.

(***) $[\![s]\!]_{\mathcal{J},\alpha} = [\varphi(s)]_\delta$, *if* $\delta(x) = \alpha(x)$ *for* $x \in FV(s)$, $\delta(x_f) = \mathcal{J}(f)$, $\delta(x_{@,\sigma}) = \mathcal{J}(@^\sigma)$

Both statements hold by a straightforward induction on the form of s.

(1) holds by (***) and Lemma 2.6(2). **(2)** holds because $[\![s\gamma]\!]_{\mathcal{J},\alpha} = [\varphi(s\gamma)]_\delta$ by (***), $= [\varphi(s)\gamma^\varphi]_\delta$ by (**), $= [\varphi(s)]_{\delta\circ\gamma^\varphi}$ by Lemma 2.6(1), which is exactly $[\![s]\!]_{\mathcal{J},\alpha\circ\gamma}$ by (***). **(3)** holds by (2): $[\![s\gamma]\!]_{\mathcal{J},\alpha} = [\![s]\!]_{\mathcal{J},\alpha\circ\gamma}$ by (2), $\sqsupseteq [\![t]\!]_{\mathcal{J},\alpha\circ\gamma} = [\![t\gamma]\!]_{\mathcal{J},\alpha}$, and similar for $\sqsupset$. **(4)** holds by a straightforward induction on the form of C (this result has no counterpart in [27]). ◀

The theory so far allows us to use weakly monotonic algebras in a *weak reduction pair.*

▶ **Theorem 3.4.** *Let a weakly monotonic algebra* $(\mathcal{A}, \mathcal{J})$ *be given such that always* $\mathcal{J}(@^\sigma) \sqsupseteq \boldsymbol{\lambda} fn.f(n)$*, and define the pair* $(\succsim, \succ)$ *by:* $s \succsim t$ *if* $[\![s]\!]_{\mathcal{J},\alpha} \sqsupseteq [\![t]\!]_{\mathcal{J},\alpha}$ *for all valuations* α*, and* $s \succ t$ *if* $[\![s]\!]_{\mathcal{J},\alpha} \sqsupset [\![t]\!]_{\mathcal{J},\alpha}$ *for all* α*. Then* $(\succsim, \succ)$ *is a weak reduction pair.*

Proof. $(\succsim, \succ)$ is a compatible combination of a quasi-ordering and a well-founded ordering by Lemma 2.3 and $\succsim$ is monotonic by Lemma 3.3(4). Also, $\succsim$ contains `beta`: for all valuations α, $[\![(\lambda x.\, s) \cdot t]\!]_{\mathcal{J},\alpha} = \mathcal{J}(@^\sigma)([\![\lambda x.\, s]\!]_{\mathcal{J},\alpha}, [\![t]\!]_{\mathcal{J},\alpha}) \sqsupseteq [\![\lambda x.\, s]\!]_{\mathcal{J},\alpha}([\![t]\!]_{\mathcal{J},\alpha})$ by assumption, which equals $[\![s]\!]_{\mathcal{J},\alpha\cup\{x\mapsto[\![t]\!]_{\mathcal{J},\alpha}\}} = [\![s]\!]_{\mathcal{J},\alpha\circ[x:=t]}$, and this equals $[\![s[x := t]]\!]_{\mathcal{J},\alpha}$ by Lemma 3.3(2). ◀

Comment: if we choose $\mathcal{J}(@^\sigma) = \boldsymbol{\lambda} fn.f(n)$, we have a system very similar to the one used for simply-typed λ-calculus (and HRSs). By not fixing the interpretation of $@^\sigma$ we have a choice, which, depending on the setting (rule removal, static dependency pairs, dynamic dependency pairs) may be essential; we will see different choices in Examples 3.5, 3.6 and 4.4.

▶ **Example 3.5.** Using the static dependency pair framework of [23] to deal with $\mathsf{shuffle}$, we obtain several sets of requirements. HORPO [17] runs into trouble with the dependency pair $\mathsf{shuffle}^\sharp(F, \mathsf{cons}(h,t)) \to \mathsf{shuffle}^\sharp(F, \mathsf{reverse}(t))$, where we need a weak reduction pair satisfying:

$$\begin{array}{rclrcl} \mathsf{shuffle}^\sharp(F, \mathsf{cons}(h,t)) & \succ & \mathsf{shuffle}^\sharp(F, \mathsf{reverse}(t)) & & & \\ \mathsf{append}(\mathsf{cons}(h,t), l) & \succsim & \mathsf{cons}(h, \mathsf{append}(t,l)) & \mathsf{append}(\mathsf{nil}, l) & \succsim & l \\ \mathsf{reverse}(\mathsf{cons}(h,t)) & \succsim & \mathsf{append}(\mathsf{reverse}(t), \mathsf{cons}(h, \mathsf{nil})) & \mathsf{reverse}(\mathsf{nil}) & \succsim & \mathsf{nil} \end{array}$$

Using Theorem 3.4, we choose the following interpretation $\mathcal{J}$ in the natural numbers:

$$\begin{array}{rclrclrcl} \mathcal{J}(\mathsf{shuffle}^\sharp) & = & \boldsymbol{\lambda} fn.n & \mathcal{J}(\mathsf{cons}) & = & \boldsymbol{\lambda} nm.m+1 & \mathcal{J}(\mathsf{nil}) & = & 0 \\ \mathcal{J}(\mathsf{reverse}) & = & \boldsymbol{\lambda} n.n & \mathcal{J}(\mathsf{append}) & = & \boldsymbol{\lambda} nm.n+m & \mathcal{J}(@^\sigma) & = & \boldsymbol{\lambda} fn.f(n) \text{ for all } \sigma \end{array}$$

Quantifying over the valuation, it suffices to show that for all $F \in \mathcal{WM}_{\mathsf{nat}\Rightarrow\mathsf{nat}}, h, t \in \mathbb{N}$: $t+1 > t,\ t+l+1 \geq t+l+1,\ t+1 \geq t+1,\ l \geq l,\ 0 \geq 0$. This is obviously the case!

▶ **Example 3.6.** For a case where we cannot choose $\mathcal{J}(@^\sigma) = \boldsymbol{\lambda} fn.f(n)$, consider $\mathsf{collapse}$:

$$\begin{array}{rclrclrcl} 0 & : & \mathsf{nat} & \mathsf{min} & : & [\mathsf{nat} \times \mathsf{nat}] \Rightarrow \mathsf{nat} & \mathsf{cons} & : & [(\mathsf{nat} \Rightarrow \mathsf{nat}) \times \mathsf{flist}] \Rightarrow \mathsf{flist} \\ \mathsf{s} & : & [\mathsf{nat}] \Rightarrow \mathsf{nat} & \mathsf{diff} & : & [\mathsf{nat} \times \mathsf{nat}] \Rightarrow \mathsf{nat} & \mathsf{build} & : & [\mathsf{nat}] \Rightarrow \mathsf{flist} \\ \mathsf{nil} & : & \mathsf{flist} & \mathsf{gcd} & : & [\mathsf{nat} \times \mathsf{nat}] \Rightarrow \mathsf{nat} & \mathsf{collapse} & : & [\mathsf{flist}] \Rightarrow \mathsf{nat} \end{array}$$

$$
\begin{array}{rclrcl}
\mathsf{min}(x,\mathsf{0}) & \to & \mathsf{0} & \mathsf{gcd}(\mathsf{s}(x),\mathsf{0}) & \to & \mathsf{s}(x) \\
\mathsf{min}(\mathsf{0},x) & \to & \mathsf{0} & \mathsf{gcd}(\mathsf{0},\mathsf{s}(x)) & \to & \mathsf{s}(x) \\
\mathsf{min}(\mathsf{s}(x),\mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{min}(x,y)) & \mathsf{gcd}(\mathsf{s}(x),\mathsf{s}(y)) & \to & \mathsf{gcd}(\mathsf{diff}(x,y),\mathsf{s}(\mathsf{min}(x,y))) \\
\mathsf{diff}(x,\mathsf{0}) & \to & x & \mathsf{build}(\mathsf{0}) & \to & \mathsf{nil} \\
\mathsf{diff}(\mathsf{0},x) & \to & x & \mathsf{build}(\mathsf{s}(x)) & \to & \mathsf{cons}(\lambda y.\,\mathsf{gcd}(y,x),\mathsf{build}(x)) \\
\mathsf{diff}(\mathsf{s}(x),\mathsf{s}(y)) & \to & \mathsf{diff}(x,y) & \mathsf{collapse}(\mathsf{nil}) & \to & \mathsf{0} \\
 & & & \mathsf{collapse}(\mathsf{cons}(F,t)) & \to & F\cdot\mathsf{collapse}(t)
\end{array}
$$

This AFS is not plain function passing, so we cannot use static dependency pairs. Using dynamic dependency pairs, HORPO runs into trouble when faced with the constraints:

$$
\begin{array}{lcllcl}
\mathsf{collapse}^\sharp(\mathsf{cons}(F,t)) & \underset{(\sim)}{\succ} & F\cdot\mathsf{collapse}(t) & & & \\
\mathsf{collapse}^\sharp(\mathsf{cons}(F,t)) & \underset{(\sim)}{\succ} & \mathsf{collapse}^\sharp(t) & l & \succsim & r \quad \text{for all rules } l\to r \text{ listed above}
\end{array}
$$

The $\underset{(\sim)}{\succ}$ relation denotes that the constraint can either be oriented with $\succsim$ or with $\succ$; to make progress, at least one of these constraints must be oriented with $\succ$. Recall that in the dynamic dependency pair approach the constraints must be satisfied with a reduction pair that also has $s\cdot t_1\cdots t_n \succsim t_i\cdot c_1\cdots c_m$ if both sides have base type, for fresh constants c_j; moreover, we must have $\mathsf{gcd}(x,y)\succsim x,y$. To guarantee this, we choose $\mathcal{J}(@^{\sigma\Rightarrow\tau}) = \boldsymbol{\lambda} fn.\max_\tau(f(n),n(\vec{0}))$, where $n(\vec{0})$ and $\max_\tau$ were defined in Example 2.4. Then $\mathcal{J}(@^{\sigma\Rightarrow\tau}) \sqsupseteq \boldsymbol{\lambda} fn.f(n)$, and if we assign $\mathcal{J}(c_j)=0_\sigma$ for $c_j:\sigma$, then $[\![s\cdot\vec{t}]\!]_{\mathcal{J},\alpha}\sqsupseteq[\![t_i\cdot\vec{c}]\!]_{\mathcal{J},\alpha}$ is indeed satisfied. Additionally, let $\mathcal{J}(\mathsf{0})=\mathcal{J}(\mathsf{nil})=0$, $\mathcal{J}(\mathsf{diff})=\mathcal{J}(\mathsf{gcd})=\boldsymbol{\lambda} nm.n+m$, $\mathcal{J}(\mathsf{s})=\mathcal{J}(\mathsf{build})=\boldsymbol{\lambda} n.3\cdot n$, $\mathcal{J}(\mathsf{min})=\boldsymbol{\lambda} nm.0$, $\mathcal{J}(\mathsf{collapse})=\boldsymbol{\lambda} n.n$, $\mathcal{J}(\mathsf{collapse}^\sharp)=\boldsymbol{\lambda} n.n+1$ and $\mathcal{J}(\mathsf{cons})=\boldsymbol{\lambda} fn.f(n)+n$.

With this interpretation, we have $l\succsim r$ for all rules. Moreover, $[\![\mathsf{collapse}^\sharp(\mathsf{cons}(F,t))]\!]_{\mathcal{J},\alpha} = 1+F(t)+t > \max(F(t),t) = [\![F\cdot\mathsf{collapse}(t)]\!]_{\mathcal{J},\alpha}$ and $[\![\mathsf{collapse}^\sharp(\mathsf{cons}(F,t))]\!]_{\mathcal{J},\alpha} = 1+F(t)+t \geq 1+t = [\![\mathsf{collapse}^\sharp(t)]\!]_{\mathcal{J},\alpha}$. As required, we can remove one dependency pair (the first one).

Strong Monotonicity To use weakly monotonic algebras in the setting of rule removal, we shall need an additional requirement: $\sqsupset$ must be monotonic. This is achieved by posing a restriction on $\mathcal{J}$: each $\mathcal{J}(f)$ should be *strongly monotonic*:

▶ **Definition 3.7** (Strongly Monotonic Functional)**.** An element f of $\mathcal{WM}_{\sigma_1\Rightarrow\ldots\Rightarrow\sigma_n\Rightarrow\iota}$ is *strongly monotonic in argument i* if for all $N_1\in\mathcal{WM}_{\sigma_1},\ldots,N_n\in\mathcal{WM}_{\sigma_n}$ and $M_i\in\mathcal{WM}_{\sigma_i}$ we have: $f(N_1,\ldots,N_i,\ldots,N_n)\sqsupset f(N_1,\ldots,M_i,\ldots,N_n)$ if $N_i\sqsupset M_i$.

For first- and second-order functions, strong monotonicity corresponds with the notion *strict* in [27]. For higher-order functions, the definition of [27] is more permissive. We have chosen to use strong monotonicity because the strictness requirement significantly complicates the theory of [27], and most common examples of higher-order systems are second-order. Strongly monotonic functionals exist for all types, e.g. $\boldsymbol{\lambda} x_1\ldots x_n.x_1(\vec{0})+\ldots+x_n(\vec{0})\in\mathcal{WM}_{\tau_1\Rightarrow\ldots\Rightarrow\tau_n\Rightarrow\iota}$.

An *extended monotonic algebra* is a weakly monotonic algebra where each $\mathcal{J}(@^\sigma)$ is strongly monotonic in its first two arguments,[3] and for $f:[\sigma_1\times\ldots\times\sigma_n]\Rightarrow\tau\in\mathcal{F}$ also $\mathcal{J}(f)$ is strongly monotonic in its first n arguments. This notion extends the corresponding definition from [9] for the first-order setting to the setting of AFSs. We obtain:

▶ **Theorem 3.8.** *Let an extended monotonic algebra $(\mathcal{A},\mathcal{J})$ be given such that always $\mathcal{J}(@^\sigma)\sqsupseteq\boldsymbol{\lambda} fn.f(n)$; the pair $(\succsim,\succ)$ from Theorem 3.4 is a strong reduction pair.*

Proof. It is a weak reduction pair by Theorem 3.4, and strongly monotonic because $[\![C[s]]\!]_{\mathcal{J},\alpha}\sqsupset[\![C[t]]\!]_{\mathcal{J},\alpha}$ for all α whenever $[\![s]\!]_{\mathcal{J},\alpha}\sqsupset[\![t]\!]_{\mathcal{J},\alpha}$ for all α (an easy induction). ◀

[3] Note that e.g. $\mathcal{J}(@^{o\Rightarrow o\Rightarrow o})$ is an element of the function space $\mathcal{WM}_{o\Rightarrow o\Rightarrow o}\Rightarrow\mathcal{WM}_o\Rightarrow\mathcal{WM}_o\Rightarrow\mathcal{WM}_o$; a function which takes *three* arguments. It need not be strongly monotonic in its 3rd argument, because we think of application as a symbol $@^{\sigma\Rightarrow\tau}:[(\sigma\Rightarrow\tau)\times\sigma]\Rightarrow\tau$ of arity *2*, where τ may be functional.

4 Higher-Order Polynomial Interpretations

It remains to be seen how to *find* suitable polynomial interpretations, preferably automatically. In this section, we will discuss the class of *higher-order polynomials over* $\mathbb{N}$, a specific subclass of the weakly monotonic functionals with $(\mathbb{N}, >, \geq)$ as a well-founded base set. In the following, we will see how suitable polynomials can be found automatically.

▶ **Definition 4.1** (Higher-Order Polynomial over $\mathbb{N}$). For a set $X = \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$ of variables, each equipped with a type, the set $Pol(X)$ of *higher-order polynomials* in X is given by the following clauses:

- if $n \in \mathbb{N}$, then $n \in Pol(X)$;
- if $p_1, p_2 \in Pol(X)$, then $p_1 + p_2 \in Pol(X)$ and $p_1 \cdot p_2 \in Pol(X)$;
- if $x_i : \tau_1 \Rightarrow \dots \Rightarrow \tau_m \Rightarrow \iota \in X$ with $\iota \in \mathcal{B}$, and $p_1 \in Pol^{\tau_1}(X), \dots, p_m \in Pol^{\tau_m}(X)$, then $x_i(p_1, \dots, p_m) \in Pol(X)$;
  - here, $Pol^{\iota}(X) = Pol(X)$ for base types ι, and $Pol^{\sigma \Rightarrow \tau}(X)$ contains functions $\boldsymbol{\lambda} y.p \in \mathcal{WM}_\sigma$ with $p \in Pol^{\tau}(X \cup \{y\})$.

We do not fix the set X. A *higher-order polynomial* is an element of any $Pol(X)$.

Noting that $\mathcal{WM}_\sigma = \mathcal{WM}_\tau$ if σ and τ have the same "form" (so are equal modulo renaming of base types), the following lemma holds for all $\iota \in \mathcal{B}$:

▶ **Lemma 4.2.** *If* $p \in Pol(\{x_1 : \sigma_1, \dots, x_n : \sigma_n\})$, *then* $\boldsymbol{\lambda} x_1 \dots x_n.p \in \mathcal{WM}_{\sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \iota}$.

Proof. It is easy to see that $+$ and $\cdot$ are weakly monotonic. Taking this into account, the lemma follows quickly with induction on the size of p, using Lemma 2.6(2). For the variable case, if $\boldsymbol{\lambda}\vec{y}.p_i \in Pol^{\tau_i}(\{\vec{x}\})$, then $p_i \in Pol(\{\vec{x}, \vec{y}\})$, so the induction hypothesis applies. ◀

Higher-order polynomials are typically represented in the form $a_1 + \dots + a_n$ (with $n \geq 0$), where each a_i is a *higher-order monomial*: an expression of the form $b \cdot c_1 \cdots c_m$, where $b \in \mathbb{N}$ and each c_i is either a base-type variable x or a function application $x(\boldsymbol{\lambda}\vec{y_1}.p_1, \dots, \boldsymbol{\lambda}\vec{y_k}.p_k)$ with all p_j higher-order polynomials again. Examples of higher-order polynomials over the natural numbers are for instance 0 and $3 + 5 \cdot x^2 \cdot y + F(37 + x)$. To find a *strongly monotonic* functional, it suffices to include, for all variables, a monomial containing only that variable:

▶ **Lemma 4.3.** *Let* $P(x_1, \dots, x_n)$ *be a higher-order polynomial of the form* $p_1(\vec{x}) + \dots + p_m(\vec{x})$, *where all* $p_i(\vec{x})$ *are higher-order monomials. Then* $\boldsymbol{\lambda}\vec{x}.P(\vec{x})$ *is strongly monotonic in argument* i *if there is some* p_j *of the form* $a \cdot x_i(\vec{b}(\vec{x}))$, *where* $a \in \mathbb{N}^+$.

Proof. Let $x_i \sqsupset x_i'$, so also $x_i \sqsupseteq x_i'$ (since $> \subseteq \geq$). Let $\vec{x} := x_1, \dots, x_i, \dots, x_l$ and $\vec{x'} := x_1, \dots, x_i', \dots, x_l$. All p_k are weakly monotonic by Lemma 4.2, so $p_k(\vec{x}) \sqsupseteq p_k(\vec{x'})$. Since $p_j(\vec{x}) \sqsupset p_j(\vec{x'})$ and $+$ is strongly monotonic, indeed $P(\vec{x}) \sqsupset P(\vec{x'})$. ◀

▶ **Example 4.4.** For rule removal on the AFS shuffle from Ex. 2.1, consider the interpretation:

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{J}(\mathsf{append}) & = & \boldsymbol{\lambda} nm.n + m & \mathcal{J}(\mathsf{cons}) & = & \boldsymbol{\lambda} nm.n + m + 3 \\
\mathcal{J}(\mathsf{reverse}) & = & \boldsymbol{\lambda} n.n + 1 & \mathcal{J}(\mathsf{nil}) & = & 0 \\
\mathcal{J}(\mathsf{shuffle}) & = & \boldsymbol{\lambda} Fn.2n + F(0) + nF(n) + 1 & \mathcal{J}(@^\sigma) & = & \boldsymbol{\lambda} fn\vec{m}.f(n, \vec{m}) + n(\vec{0}) \ (**)
\end{array}$$

(**) Here, $n(\vec{0})$ is the "lowest value" function from Ex. 2.4. With this interpretation, which is a strongly monotonic polynomial interpretation by Lemma 4.3, all rules are oriented with $\succsim$, and the two shuffle rules and the reverse(nil) one even with $\succ$. Only for the main shuffle rule this is non-trivial to see; here we have the constraint: $F(h + t + 3) + tF(h + t + 3) + F(h + t + 3) + [h + hF(h + t + 3) + 3F(h + t + 3)] + 2 > F(h) + tF(t + 1) + F(t + 1)$. This holds by weak monotonicity of F: since $h + t + 3 \geq h$ always holds, we must have $F(h + t + 3) \geq F(h)$ as well, and similarly we see that $tF(h + t + 3) \geq tF(t + 1)$ and $F(h + t + 3) \geq F(t + 1)$.

5 Automation

To demonstrate that the approach is automatable, we have made a proof-of-concept implementation of polynomial interpretations in the higher-order termination tool WANDA. The implementation only tries simple parametric shapes, does not use heuristics, and is limited to second-order AFSs – a limitation which excludes but 5 out of the 156 higher-order benchmarks in the current *termination problem database (TPDB)*,[4] as the class of second-order systems is very common.[5] Even with this minimal implementation, the combination of polynomial interpretations with dependency pairs can handle about 75% of the TPDB.

To find polynomial interpretations automatically, WANDA uses the following steps:

1. assign every function symbol a higher-order polynomial with *parameters* as coefficients;
2. for all requirements $l \mathrel{{}_{(}\succsim_{)}} r$ and $l \succsim r$, calculate $[\![l]\!]_{\mathcal{J},\alpha}$ and $[\![r]\!]_{\mathcal{J},\alpha}$ as a function on parameters and variables – this gives constraints $P_i \mathrel{{}_{(}\geq_{)}} Q_i$ and $P_i \geq Q_i$;
3. introduce a parameter o_i for all constraints of the form $P_i \mathrel{{}_{(}\geq_{)}} Q_i$, and replace these constraints by $P_i \geq Q_i + o_i$; if we also introduce the constraints $o_1 + \ldots + o_n \geq 1$ then, when all constraints are satisfied, at least one ${}_{(}\geq_{)}$ constraint is strictly oriented;
4. simplify the constraints until they no longer contain variables;
5. impose maximum values on the search space of the parameters and use a non-linear constraint solver to find a solution for the constraints.

These steps are detailed below, with an AFS rule for the function `map` as a running example.

5.1 Choosing Parametric Polynomial Interpretations

The module for polynomial interpretations in WANDA is called in three contexts: rule removal, the dynamic dependency pair framework and the static dependency pair framework. In the first case, function interpretations must be strongly monotonic, in the second case they have to satisfy a subterm property, and in the third there are no further restrictions.

To start, every function symbol $f : [\sigma_1 \times \ldots \times \sigma_n] \Rightarrow \sigma_{n+1} \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota \in \mathcal{F}$ is assigned a function of the form $\boldsymbol{\lambda} x_1 \ldots x_m . p_1 + p_2 + a$, where a is a parameter and:

- p_1 has the form $a_1 \cdot x_1(0, \ldots, 0) + \ldots + a_m \cdot x_m(0, \ldots, 0)$, where the a_i are parameters (this is well-typed because we work in a second-order system);
  - in the rule removal setting, we add requirements: $a_1 \geq 1, \ldots, a_n \geq 1$;
  - in the dynamic dependency pairs setting, we add requirements: $a_{n+1} \geq 1, \ldots, a_m \geq 1$.
- $p_2 = q_1 + \ldots + q_k$, where each q_j has the form $c_j \cdot x_{i_1} \cdots x_{i_k} \cdot x_j(x_{i_1}, \ldots, x_{i_k}) + d_j \cdot x_j(x_{i_1}, \ldots, x_{i_k})$, with c_j, d_j parameters, the x_{i_l} first-order variables, and x_j a higher-order variable; every combination of a higher-order variable with first-order variables occurs.[6]

We must also choose an interpretation of $@^\sigma$ for all types. Rather than using a parametric interpretation, we observe that application occurs mostly on the right-hand side of constraints. There, we often have (sub-)terms $F \cdot s_1 \cdots s_n$ with F a free variable; on the left-hand side,

[4] See `http://termination-portal.org/wiki/TPDB` for details on this standard database.

[5] The restriction to second-order systems is not essential, but it makes the code easier in a number of places: we can avoid representing function-polynomials $\boldsymbol{\lambda}\vec{x}.P(\vec{x})$, stick to simple interpretation shapes, and we do not have max in the left-hand side of constraints. Mostly, the restriction is present because of the low number of available benchmarks of order 3 or higher, which makes it hard to select suitable interpretation shapes, and not initially worth the added implementation effort.

[6] In case the constraint solver does not find a solution for this interpretation shape, WANDA additionally includes non-linear monomials $c_{i,j} \cdot x_i \cdot x_j$ (where $i < j$) without functional variables in the parametric higher-order polynomials and tries again. In general, here one can use arbitrary parametric polynomials.

such subterms do not occur, nor can we have applications headed by an abstraction or bound variable (in a second-order system, bound variables have base type). Only applications of the form $f(s_1,\dots,s_n)\cdot s_{n+1}\cdots s_m$ occur on the left; since function symbols usually have a base type as output type, this is a rare situation. Thus, we fix the interpretation of $@^\sigma$ for all types to be as small as possible. Note that we must have $\mathcal{J}(@^\sigma) \sqsupseteq \boldsymbol{\lambda} fn.f(n)$ by Theorem 3.4, and $\mathcal{J}(@^\sigma)$ may have to be strongly monotonic, or satisfy a subterm property.

- in the rule removal setting, $\mathcal{J}(@^\sigma) = \boldsymbol{\lambda} fn\vec{m}.f(n,\vec{m}) + n(\vec{0})$;
- in the dynamic dependency pairs setting, $\mathcal{J}(@^\sigma) = \boldsymbol{\lambda} fn\vec{m}.\max(f(n,\vec{m}), n(\vec{0}))$;
- in the static dependency pairs setting, $\mathcal{J}(@^\sigma) = \boldsymbol{\lambda} fn\vec{m}.f(n,\vec{m})$.

In the rule removal setting this choice together with the constraints on the parameters guarantees that all $\mathcal{J}(f)$ are strongly monotonic in the arguments required by the definition of an extended monotonic algebra and Theorem 3.8. In the dynamic dependency pairs setting, we obtain the required subterm property as demonstrated in Example 3.6. Moreover, in this setting always $[\![f(s_1,\dots,s_n)\cdot s_{n+1}\cdots s_m]\!]_{\mathcal{J},\alpha} = \max(\mathcal{J}(f)([\![s_1]\!]_{\mathcal{J},\alpha},\dots,[\![s_n]\!]_{\mathcal{J},\alpha},[\![s_{n+1}]\!]_{\mathcal{J},\alpha},\dots,$ $[\![s_m]\!]_{\mathcal{J},\alpha}), [\![s_{n+1}]\!]_{\mathcal{J},\alpha}(\vec{0}),\dots,[\![s_m]\!]_{\mathcal{J},\alpha}(\vec{0})) = \mathcal{J}(f)([\![s_1]\!]_{\mathcal{J},\alpha},\dots,[\![s_m]\!]_{\mathcal{J},\alpha})$ by the restriction on the parameters. Thus, although we now also need to deal with the max-operator, it will only ever occur on the right-hand side of a constraint! Since this avoids the need for conditional constraints as used in [11] (without losing any power), it both simplifies the automation and creates smaller constraints.

From these parametric higher-order polynomials, we calculate the interpretations of terms, and simplify the resulting higher-order polynomials into a sum of monomials. For the constraints $l \mathrel{_{(}\succsim_{)}} r$, in general we use constraints $[\![l]\!]_{\mathcal{J},\alpha} \geq [\![r]\!]_{\mathcal{J},\alpha} + o$ for some fresh *bit* o (a parameter whose value ranges over $\{0,1\}$), and require that the sum of these bits is positive.

▶ **Example 5.1** (Running Example). To demonstrate the technique, consider rule removal on the recursive rule of the common map example, which gives the constraint $\mathsf{map}(F,\mathsf{cons}(h,t)) \mathrel{_{(}\succsim_{)}}$ $\mathsf{cons}(F\cdot h,\mathsf{map}(F,t))$. We assign: $\mathcal{J}(\mathsf{cons}) = \boldsymbol{\lambda} nm.a_1\cdot n + a_2\cdot m + a_3$ and $\mathcal{J}(\mathsf{map}) = \boldsymbol{\lambda} fn.a_4\cdot f(0) + a_5\cdot n + a_6\cdot n\cdot f(n) + a_7$.[7] This leads to the following constraints:

- $a_1,a_2,a_4,a_5 \geq 1$, $o_1 \geq 1$ (we could also immediately replace o_1 by 1).
- $a_7 + a_3\cdot a_5 + a_1\cdot a_5\cdot h + a_2\cdot a_5\cdot t + a_4\cdot F(0) + a_1\cdot a_6\cdot h\cdot F(a_1\cdot h + a_2\cdot t + a_3) + a_2\cdot a_6\cdot t\cdot F(a_1\cdot h + a_2\cdot t + a_3) + a_3\cdot a_6\cdot F(a_1\cdot h + a_2\cdot t + a_3) \geq a_3 + a_2\cdot a_7 + o_1 + a_1\cdot h + a_2\cdot a_5\cdot t + a_2\cdot a_4\cdot F(0) + a_1\cdot F(h) + a_6\cdot t\cdot F(t)$

5.2 Simplifying Polynomial Requirements

We obtain requirements that contain variables as well as parameters; they should be read as "there exist a_i, o_k such that for all h,t,F the inequalities hold". To avoid dealing with claims over all possible numbers or functions we simplify the requirements until they contain no more variables. To a large extent, these simplifications correspond to the ones used with automations of polynomial interpretations for first-order rewriting [6], but higher-order variables in function application present an extra difficulty. To deal with application of higher-order variables, we will use Lemma 5.2:

▶ **Lemma 5.2.** *Let F be a weakly monotonic functional and all p,q,p_i,q_i,s_i,r_i polynomials.*

1. $F(r_1,\dots,r_k)\cdot p \geq F(s_1,\dots,s_k)\cdot q$ *if* $r_1 \geq s_1,\dots,r_k \geq s_k, p \geq q$.

2. $r_1\cdot p_1 + \dots + r_n\cdot p_n \geq s_1\cdot q_1 + \dots + s_m\cdot q_m$ *if there are* $e_{i,j}$ *for* $1 \leq i \leq n, 1 \leq j \leq m$ *with:*

  a. *for all i:* $r_i \geq e_{i,1} + \dots + e_{i,m}$*;*

[7] To ease presentation, in contrast to WANDA here we do not use an addend $a_i\cdot f(n)$ for map.

b. *for all j: $e_{1,j} + \ldots + e_{n,j} \geq s_j$;*
c. *either $e_{i,j} = 0$ or $p_i \geq q_j$.*

Proof. **(1)** holds by weak monotonicity of F. As for **(2)**, $r_1 \cdot p_1 + \ldots + r_n \cdot p_n \geq \sum_{i=1}^{n} \sum_{j=1}^{m} e_{i,j} \cdot p_i$ by **(a)**, and since $e_{i,j} = 0$ whenever not $p_i \geq q_j$ by **(c)**, $\sum_{i=1}^{n} \sum_{j=1}^{m} e_{i,j} \cdot p_i \geq \sum_{i=1}^{n} \sum_{j=1}^{m} e_{i,j} \cdot q_j = \sum_{j=1}^{m} \sum_{i=1}^{n} e_{i,j} \cdot q_j$. Using **(b)**, $\sum_{j=1}^{m} \sum_{i=1}^{n} e_{i,j} \cdot q_j \geq \sum_{j=1}^{m} s_j \cdot q_j$ as required. ◀

Lemma 5.2, together with some observations used in the first-order case, supplies the theory we need to simplify the requirements to constraints which do not contain any variables. Here, a "component" of a monomial $a_1 \cdots a_n$ is any of the a_i (but p is not a component of $F(p)$).

1. Do standard simplifications on the constraints, for instance replacing $3 \cdot F(n) \geq F(n) + a_1 \cdot n$ by $2 \cdot F(n) \geq a_1 \cdot n$ and $p + B \cdot p$ by $(B+1) \cdot p$ if B is a *known* constant, and removing monomials $0 \cdot p$. Remove constraints $p \geq 0$ and $p \geq p$ which always hold.
2. Split constraints $0 \geq p_1 + \ldots + p_n$ into the n constraints $0 \geq p_i$. Remove constraints $0 \geq a$ where a is a single parameter, and replace a by 0 everywhere else.
   This is valid because, in the natural numbers, $0 \geq a$ implies $a = 0$, and $0 + \ldots + 0 = 0$.
3. Replace constraints $P \geq Q[\max(r,s)]$ by the two constraints $P \geq Q[r]$ and $P \geq Q[s]$.
   This is valid because for any valuation $Q[\max(r,s)]$ equals $Q[r]$ or $Q[s]$.
4. Given a constraint $p_1 + \ldots + p_n \geq p_{n+1} + \ldots + p_m$ where some, but not all, of the monomials p_i contain a component x or $x(\vec{q})$ for some fixed variable x, let A contain the indices i of those monomials p_i which have x or $x(\vec{q})$. Replace the constraint by the two constraints $\sum_{i \in A, i \leq n} p_i \geq \sum_{i \in A, i > n} p_i$ and $\sum_{i \notin A, i \leq n} p_i \geq \sum_{i \notin A, i > n} p_i$. For example, splitting on n, the constraint $3 \cdot n \cdot m + a_2 \cdot F(a_3 \cdot n + a_4) \geq 2 + a_7 \cdot m + F(n)$ is split into $3 \cdot n \cdot m \geq 0$ and $a_2 \cdot F(a_3 \cdot n + a_4) \geq 2 + a_7 \cdot m + F(n)$; subsequently, splitting on F, the latter is split into $a_2 \cdot F(a_3 \cdot n + a_4) \geq F(n)$ and $0 \geq 2 + a_7 \cdot m$.
   This is valid because $p_1 + p_2 \geq q_1 + q_2$ certainly holds if $p_1 \geq q_1$ and $p_2 \geq q_2$.
5. If all non-zero monomials on either side of a constraint have a component x, "divide out" x. For example, replace the constraint $a_1 \cdot n + n \cdot n \cdot f(a_3, n) \geq n + a_3 \cdot n$ by $a_1 + n \cdot f(a_3, n) \geq 1 + a_3$, and replace $0 \geq a_5 \cdot m$ by $0 \geq a_5$.
   This is valid because $p \cdot n \geq q \cdot n$ holds if $p \geq q$ (cf. the absolute positiveness criterion [15]).
6. Replace a constraint $s \cdot x_1(p_{1,1}, \ldots, p_{1,k_1}) \cdots x_n(p_{n,1}, \ldots, p_{n,k_n}) \geq s \cdot x_1(q_{1,1}, \ldots, q_{1,k_1}) \cdots x_n(q_{n,1}, \ldots, q_{n,k_n})$ by the constraints $s \cdot p_{i,j} \geq s \cdot q_{i,j}$ for all i, j.
   This is valid by Lemma 5.2(1) and case analysis whether $s = 0$ or not.
7. Let $p_1, \ldots, p_n, q_1, \ldots, q_m$ be monomials of the form $x_1(\vec{r_1}), \ldots, x_k(\vec{r_k})$, for fixed $x_1, \ldots, x_k$. Replace a constraint $r_1 \cdot p_1 + \ldots + r_n \cdot p_n \geq s_1 \cdot q_1 + \ldots + s_m \cdot q_m$ with $n, m \geq 1$ by the following constraints, where the $e_{i,j}$ are fresh parameters:
   for $1 \leq i \leq n$: $r_i \geq e_{i,1} + \ldots + e_{i,m}$
   for $1 \leq j \leq m$: $e_{1,j} + \ldots + e_{n,j} \geq s_j$
   for $1 \leq i \leq n,\ 1 \leq j \leq m$: $e_{i,j} \cdot p_i \geq e_{i,j} \cdot q_j$ (which can be handled with clause 6)
   This is valid by Lemma 5.2(2).[8]

It is easy to see that while a constraint still has variables in it, we can apply clauses to simplify or split it (taking into account that max does not appear in the left-hand side of a constraint), and that the clauses also terminate on a system without variables. These simplifications are not complete: for example, a universally valid constraint $F(n) \cdot n \geq F(1) \cdot n$ is split into constraints $n \geq n$ (which holds), and $n \geq 1$ (which does not).

[8] In the cases where $n = 1$ or $m = 1$, some of these parameters are unnecessary; for instance, if $n = 1$, we can safely fix $e_{1,j} = s_j$ for all j. Our actual implementation uses a few of such special-case optimisations.

▶ **Example 5.3.** Let us simplify the constraints from Example 5.1. First, using clause 4 to group monomials by their variables, we obtain:

- $a_1, a_2, a_4, a_5, o_1 \geq 1$
- $a_7 + a_3 \cdot a_5 \geq a_3 + a_2 \cdot a_7 + o_1$
- $a_1 \cdot a_5 \cdot h \geq a_1 \cdot h$
- $a_2 \cdot a_5 \cdot t \geq a_2 \cdot a_5 \cdot t$
- $a_4 \cdot F(0) + a_3 \cdot a_6 \cdot F(a_1 \cdot h + a_2 \cdot t + a_3) \geq a_2 \cdot a_4 \cdot F(0) + a_1 \cdot F(h)$
- $a_1 \cdot a_6 \cdot h \cdot F(a_1 \cdot h + a_2 \cdot t + a_3) \geq 0$
- $a_2 \cdot a_6 \cdot t \cdot F(a_1 \cdot h + a_2 \cdot t + a_3) \geq a_6 \cdot t \cdot F(t)$

The 4th and 6th requirements are trivial and can be removed with clause 1. After dividing away the non-functional variables using clause 5 we have the following constraints left:

- $a_1, a_2, a_4, a_5, o_1 \geq 1$
- $a_7 + a_3 \cdot a_5 \geq a_3 + a_2 \cdot a_7 + o_1$
- $a_1 \cdot a_5 \geq a_1$
- $a_4 \cdot F(0) + a_3 \cdot a_6 \cdot F(a_1 \cdot h + a_2 \cdot t + a_3) \geq a_2 \cdot a_4 \cdot F(0) + a_1 \cdot F(h)$
- $a_2 \cdot a_6 \cdot F(a_1 \cdot h + a_2 \cdot t + a_3) \geq a_6 \cdot F(t)$

The first three are completely simplified. Clauses 7 and 6 replace the last two constraints by:

- $a_4 \geq e_{1,1} + e_{1,2}, \quad a_3 \cdot a_6 \geq e_{2,1} + e_{2,2}, \quad e_{1,1} + e_{2,1} \geq a_2 \cdot a_4, \quad e_{1,2} + e_{2,2} \geq a_1$
- $e_{1,1} \cdot 0 \geq e_{1,1} \cdot 0, \quad e_{1,2} \cdot 0 \geq e_{1,2} \cdot h$
- $e_{2,1} \cdot (a_1 \cdot h + a_2 \cdot t + a_3) \geq e_{2,1} \cdot 0, \quad e_{2,2} \cdot (a_1 \cdot h + a_2 \cdot t + a_3) \geq e_{2,2} \cdot h$
- $a_2 \cdot a_6 \geq k_{1,1}, \quad k_{1,1} \geq a_6, \quad k_{1,1} \cdot (a_1 \cdot h + a_2 \cdot t + a_3) \geq k_{1,1} \cdot t$

Using clauses 1, 4 and 5, we can simplify the constraints further, and obtain:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
a_1, a_2, a_4, a_5, o_1 & \geq & 1 & a_3 \cdot a_6 & \geq & e_{2,1} + e_{2,2} & a_2 \cdot a_6 & \geq & k_{1,1} \\
a_7 + a_3 \cdot a_5 & \geq & a_3 + a_2 \cdot a_7 + o_1 & e_{1,1} + e_{2,1} & \geq & a_2 \cdot a_4 & k_{1,1} & \geq & a_6 \\
a_1 \cdot a_5 & \geq & a_1 & e_{2,2} & \geq & a_1 & k_{1,1} \cdot a_2 & \geq & k_{1,1} \\
a_4 & \geq & e_{1,1} & e_{2,2} \cdot a_1 & \geq & e_{2,2} & & &
\end{array}$$

Thus, using a handful of clauses, the requirements are simplified to a number of constraints with parameters over the natural numbers. In the actual WANDA implementation a few small optimisations are used; for example, some simplifications are combined, and if $\max(r, s)$ occurs more than once in the same polynomial, all occurrences are replaced by r or s at the same time. However, these optimisations make no fundamental difference to the method.

After imposing bounds on the search space, we can solve the resulting non-linear constraints using standard SAT- [10] or SMT-based [3] techniques (WANDA uses a SAT encoding similar to [10] with the solver MiniSAT [7] as back-end). If the problem is satisfiable, the solver returns values for all parameters, so it is easy to see which requirements have been oriented with $>$. For map, the solver could provide for example the solution $a_7 = e_{2,1} = 0$, $a_1 = a_2 = a_3 = a_4 = a_6 = e_{1,1} = e_{2,2} = k_{1,1} = o_1 = 1$, and $a_5 = 2$. This results in the interpretation $\mathcal{J}(\mathsf{cons}) = \lambda nm.n + m + 1$ and $\mathcal{J}(\mathsf{map}) = \lambda fn.f(0) + 2 \cdot n + n \cdot f(n)$.

6 Experiments

For an empirical evaluation of our contributions, we conducted a number of experiments with our implementation in WANDA using an Intel Xeon 5140 CPU with four cores at 2.33 GHz (cf. also `http://aprove.informatik.rwth-aachen.de/eval/HOPOLO/` for details on the evaluation). As underlying benchmark set, we used the 156 examples from the higher-order category of the TPDB version 8.0.1 together with Examples 2.1 and 3.6. WANDA invokes the SAT solver MiniSAT [7] and the first-order termination prover AProVE [14] as back-ends.

As in the Termination Competition, we imposed a 60 second timeout per example.

The module for polynomial interpretation is called potentially twice with different polynomial shapes, as described at the start of Section 5.1 (cf. Footnote 6). The search space for the parameters is $\{0, \dots, 3\}$. Our first experiment is designed to analyse the impact of polynomial interpretations coupled with a higher-order dependency pair framework.

Configuration	YES	NO	MAYBE	TIMEOUT	Avg. time
WANDA full	124	9	23	2	3.19 s
WANDA no poly	119	9	30	0	2.40 s
WANDA no horpo	118	9	28	3	3.59 s

Figure 1 *Experimental results of full WANDA with and without polynomials or horpo*

Fig. 1 shows the results of *WANDA full*, which includes both polynomial interpretations and HORPO, the other main class of orderings implemented by WANDA (other than that, WANDA only uses the subterm criterion as an ordering-based technique). They are compared to versions of WANDA where either polynomials or HORPO are disabled. Although WANDA already scored highest in the Termination Competition of 2011, adding the contributions of this paper gives an additional 5 examples on the benchmark set. It is interesting to note that even without HORPO, WANDA with polynomials can still show termination of 118 examples.

Using the contributions of [12], WANDA delegates the first-order part of a higher-order rewrite system to the first-order termination tool AProVE, where it is commonplace to use polynomial interpretations. The setup of our second experiment deals with the impact of higher-order polynomial interpretations if WANDA does not use a first-order tool.

Configuration	YES	NO	MAYBE	TIMEOUT	Avg. time
WANDA no [12] full	118	9	29	2	2.32 s
WANDA no [12] no poly	107	9	42	0	1.09 s
WANDA no [12] no horpo	111	9	35	3	2.89 s

Figure 2 *Experimental results of WANDA without first-order back-end*

Fig. 2 juxtaposes the results of WANDA without the first-order prover AProVE in three configurations. We see that if we disable the first-order back-end, the increase in power by polynomial interpretations goes up from 5 examples in the first experiment to 11 examples in the second. Thus, the gain of using a first-order tool can at least partially be compensated by using native higher-order polynomial interpretations.

Our third experiment investigates the impact of higher-order polynomial interpretations if no dependency pairs are used (which also excludes first-order termination tools). Here we compare to the version of HORPO implemented in WANDA.

Configuration	YES	NO	MAYBE	TIMEOUT	Avg. time
Rule Removal both	76	9	70	3	3.60 s
Rule Removal horpo	69	9	80	0	1.01 s
Rule Removal poly	47	9	97	5	3.64 s

Figure 3 *Experimental results of WANDA with rule removal (and without dependency pairs)*

Using just rule removal, HORPO clearly trumps polynomial interpretations. However, in part this may be due to the limited choice in interpretation shapes this first implementation of polynomial interpretations supports.

Discussion Analysing the termination problem database, it is perhaps not surprising that the gain from using polynomial interpretations in the first experiment is not larger: the majority of the benchmarks which WANDA cannot already handle is non-terminating, or not known to be terminating (for example, state-of-the-art first-order tools cannot prove termination of the first-order part). For others, type-conscious methods such as *accessibility* (see e.g. [2]) are required; the method described in this paper ignores differences in base types. For cases where polynomial interpretations are needed, but only for the (truly) first-order part, passing this first-order part [12] to a modern first-order tool already suffices – as is evident by comparing the numbers in the first and second experiments. With higher-order polynomial interpretations, we have gained three out of the remaining seven benchmarks.

7 Conclusion

In this paper, we have extended the termination method of weakly monotonic algebras to the class of AFSs, simplifying definitions and adding the theory to use algebras with rule removal and dependency pairs; some efforts towards this were previously made in [20], but only for the setting of dynamic dependency pairs. Then, we introduced the class of higher-order polynomial interpretations, and discussed how suitable interpretations can be found automatically. The implementation of polynomial interpretations increases the power of WANDA by a respectable five benchmarks, including the two examples in this paper.

Thus, weakly monotonic algebras form an elegant method for proving termination by hand and, as demonstrated by the implementation in WANDA and the results of the experiments, a feasible automatable termination method as well.

Future Work We have by no means reached the limit of what can be achieved with this technique: we might consider different interpretation shapes, possibly coupled with heuristics to determine a suitable shape. Or we may go beyond polynomials; we could for instance use max in function interpretations as done in e.g. [11], or (for a truly higher-order alternative), use repeated function application; this leads to interpretations like $\boldsymbol{\lambda} nmf.\max(m, f^n(m))$.[9]

Another alley to explore is to combine polynomial interpretations with type interpretations: rather than collapsing all base types into one, we might *translate* them, e.g. mapping a base type funclist to o⇒o. WANDA already does this in very specific cases, and one could simultaneously search for polynomial interpretations and for a type interpretation – this could parallel the search for type orderings in implementations of the recursive path ordering [2].

Moreover, in the first-order world, there are many more applications of monotonic algebras, e.g. matrix, arctic, rational, real and integer interpretations… There is no obvious reason why these methods cannot be lifted to the higher-order case as well!

References

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

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

3 C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. SAT modulo linear arithmetic for solving polynomial constraints. *Journal of Automated Reasoning*, 48(1):107–131, 2012.

4 C. Borralleras and A. Rubio. THOR – a higher-order termination tool. `http://www.lsi.upc.edu/~albert/term.html`.

5 C. Borralleras and A. Rubio. A monotonic higher-order semantic path ordering. In *Proc. LPAR 2001*, LNAI 2250, pages 531–547, 2001.

[9] The max is essential in this interpretation because $\boldsymbol{\lambda} nmf.f^n(m)$ is not weakly monotonic. A case analysis whether $m \geq f(m)$ or $f(m) \geq m$ shows that $\boldsymbol{\lambda} nmf.\max(m, f^n(m))$ *is* weakly monotonic.

6 E. Contejean, C. Marché, A. P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325–363, 2005.

7 N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. SAT 2003*, LNCS 2919, pages 502–518, 2004.

8 J. Endrullis. Jambox. `http://joerg.endrullis.de/`.

9 J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning*, 40(2-3):195–220, 2008.

10 C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *Proc. SAT 2007*, LNCS 4501, pages 340–354, 2007.

11 C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. Maximal termination. In *Proc. RTA 2008*, LCNS 5117, pages 110–125, 2008.

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

13 C. Fuhs and C. Kop. Polynomial interpretations for higher-order rewriting. Technical Report `arXiv:1203.5754 [cs.LO]`, 2012. `http://arxiv.org/abs/1203.5754`.

14 J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. IJCAR 2006*, LNAI 4130, pages 281–286, 2006.

15 H. Hong and D. Jakuš. Testing positiveness of polynomials. *Journal of Automated Reasoning*, 21(1):23–38, 1998.

16 J.-P. Jouannaud and M. Okada. A computation model for executable higher-order algebraic specification languages. In *Proc. LICS 1991*, pages 350–361, 1991.

17 J.-P. Jouannaud and A. Rubio. Polymorphic higher-order recursive path orderings. *Journal of the ACM*, 54(1):1–48, 2007.

18 C. Kop. WANDA – a higher order termination tool. `http://few.vu.nl/~kop/code.html`.

19 C. Kop. Simplifying algebraic functional systems. In *Proc. CAI 2011*, LNCS 6742, pages 201–215, 2011.

20 C. Kop and F. van Raamsdonk. Higher order dependency pairs for algebraic functional systems. In *Proc. RTA 2011*, LIPIcs 10, pages 203–218, 2011.

21 C. Kop and F. van Raamsdonk. Dynamic dependency pairs for algebraic functional systems. *Logical Methods in Computer Science*, 2012. Special Issue of the 22nd International Conference on Rewriting Techniques and Applications (RTA 2011). To appear.

22 M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In *Proc. RTA 2009*, LNCS 5595, pages 295–304, 2009.

23 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.

24 D. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.

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

26 J. van de Pol. Termination proofs for higher-order rewrite systems. In *Proc. HOA 1993*, LNCS 816, pages 305–325, 1994.

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

28 M. Sakai, Y. Watanabe, and T. Sakabe. An extension of the dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E84-D(8):1025–1032, 2001.

29 Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

30 Wiki. Termination portal. `http://www.termination-portal.org/`.