Transposing Termination Properties in Higher Order Rewriting

Cynthia Kop
Department of Computer Science
VU University Amsterdam, the Netherlands.
kop@few.vu.nl

In higher order term rewriting, distinguished from first order term rewriting by the presence of bound variables and often also a type discipline, a plethora of different frameworks is used. Consequently, proofs of termination properties derived in one framework often have to be redone for others. In this paper we formulate a generalised framework for higher order rewriting that facilitates an easy transposition of termination properties between different frameworks. For the main higher order rewrite formalisms currently in use, we provide translations to and from this central generalised framework. These translations preserve the termination property. Care is taken to make this intermediate framework on the one hand general enough, on the other hand simple enough; the one step rewriting relation is decidable.

1 Introduction

The last few years have seen a rise in the interest in higher order rewriting, especially in the field of termination. While this area is still far behind its first order counterpart, various techniques for proving termination have been developed, such as higher order path orderings [5, 2] and dependency pairs [16, 11]. Since 2010 the annual termination competition [17] has a category for higher order termination.

However, a persistent problem is the lack of a fixed standard. There are several important formalisms, each dealing with the higher order aspect in a different way, along with many variations and restrictions. Because of the differences in what is and is not allowed, results in one formalism do not trivially, or not at all, carry over to another. As such it is difficult to reuse results in a slightly different context, which necessitates a lot of double work. Consider for example the original HORPO, first defined for Algebraic Functional Systems in [5] and adapted for Pattern HRSs in [15].

This became in particular clear in our endeavour to develop a tool for proving and disproving termination. The question what sort of formalism to support is essential from the start. Should we choose to implement a tool for Nipkow's Pattern HRSs, then we cannot handle a system with a rule $\lambda x.Z\ x \Rightarrow Z$, and the tool would give a false positive on a system with two rules $f\ 0 \Rightarrow g\ (\lambda x.0)$ and $g\ F \Rightarrow F\ (f\ 0)$ – both of which are valid programs in Jouannaud's Algebraic Functional System formalism. On the other hand, a tool for AFSs could not handle a system with rules like $f\ (\lambda x.g\ (F\ x)) \Rightarrow F\ 0$, and would give a false negative on the system suggested above. And even if we just accept that we cannot support everything, the price for any choice is substantial: for example, the latest version of a recursive path ordering [2] is only developed for AFSs, while static dependency pairs [11] have only been defined for HRSs.

To improve this situation, we propose to embed all the common forms of (higher order) rewriting into a general framework. This serves the double goal of facilitating the translation of results from one formalism to another, as well as providing a basis for termination tools. Rather than choosing one existing formalism over the others, we will define a new formalism which encompasses all the common higher order formats with binders. This proposed formalism builds on the common ground of the usual formalisms, includes polymorphism and is not too abstract to obtain real (termination) results, such as an extension of the Computability Path Ordering [5]. Unlike van Oostrom's HORSs [14], which have a comparable goal but are defined on a more abstract level, the formalism stays close enough to common formalisms to extend non-abstract results.[1]

Note that our primary aim is not to introduce yet another term rewriting formalism; rather, we show how programs in the existing formalisms can be transformed and made to follow a general format, so it is straightforward how to transfer results from one style of rewriting to another.

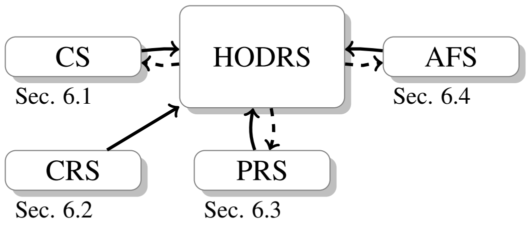

Figure 1. Transformations between higher order formalisms. A heavy arrow indicates a full embedding from one formalism into another. A dashed arrow indicates a partial embedding. The central HODRS formalism is the new framework defined in this paper.

This paper focusses on termination and thus ignores other interesting topics such as confluence in the formalism. In addition, Nipkow's HRSs with unrestricted left-hand sides cannot be represented in the proposed formalism. This is because for any system containing HRSs it will be hard to obtain strong results, since the rewrite relation for an HRS in general lacks desirable properties such as decidability of the one-step rewrite relation. The common restriction to *pattern HRSs*, however, is covered.

2 Definition

In this section we define our formalism, called *Higher Order Decidable Rewrite Systems*. They are polymorphic applicative systems, using meta-variables for matching like in Aczel's Contraction Schemes [1], Klop's Combinatory Reduction Systems [8] and Khasidashvili's Expression Reduction Systems [7] (which are very similar to CRSs, but were introduced independently).

Types We assume a set of *type constructors* $\mathscr{B}$ and a set of *type variables* $\mathscr{A}$. Each type constructor b has a fixed arity, notation $ar(b) = n$ with n a natural number; at least one has arity 0. A *polymorphic type* is an expression over $\mathscr{B}$ and $\mathscr{A}$ built according to the following grammar:

$$\mathscr{T} = \alpha \mid b(\mathscr{T}^n) \mid \mathscr{T} \to \mathscr{T} \quad (\alpha \in \mathscr{A},\ b \in \mathscr{B},\ ar(b) = n)$$

A *monomorphic* type does not contain type variables. A type of the form $\sigma_1 \to \sigma_2$ is *functional*, and a type $b(\sigma_1,\dots,\sigma_n)$ is a *data type*. A type $b()$ with $ar(b) = 0$ is denoted as just b. Types are written as $\sigma,\tau,\rho,\dots$ and type variables as $\alpha_1,\alpha_2,\dots$. The $\to$ associates to the right.

Example 1. Some example types: $\mathtt{nat}$, $\mathtt{nat} \to \mathtt{bool}$, $\mathtt{list}(\mathtt{nat})$ and (non-monomorphic) $\alpha \to \mathtt{list}(\alpha)$.

We say $\sigma \geq \tau$ if τ can be obtained from σ by substituting types for type variables. For example, $\alpha \geq \alpha \geq \alpha \to \beta \geq \mathtt{nat} \to \mathtt{nat}$, but not $\alpha \to \alpha \geq \mathtt{nat} \to \mathtt{real}$. A *type declaration* is an expression of the form $(\sigma_1 \times \ldots \times \sigma_n) \longrightarrow \tau$ with $\sigma_1,\dots,\sigma_n,\tau \in \mathscr{T}$; such a declaration is said to have arity n. Type declarations are not types, but are used to type meta-variables. A type declaration $() \longrightarrow \tau$ is usually just denoted by τ. We also use $\geq$ to compare two type declarations.

[1] The more abstract level in van Oostrom's HORSs is given by an underlying substitution calculus which generally speaking consists of a version of λ-calculus with β-reduction.

Terms and Meta-terms A *meta-term* is an expression s over a set $\mathcal{F}$ of typed constants (also known as *function symbols*), a set $\mathcal{V}$ of variables and a set $\mathcal{M}$ of *meta-variables*, each meta-variable equipped with an arity n, such that $s : \sigma$ can be derived for some type σ with the following rules:

`(var)`	$x_\tau : \tau$	if $x \in \mathcal{V}$ and τ a type
`(fun)`	$f_\tau : \tau$	if $f_\sigma \in \mathcal{F}$ and $\sigma \geq \tau$
`(abs)`	$\lambda x_\sigma.s : \sigma \rightarrow \tau$	if $x \in \mathcal{V}$, σ a type and $s : \tau$
`(app)`	$s \cdot t : \tau$	if $s : \sigma \rightarrow \tau$ and $t : \sigma$
`(meta)`	$Z_{(\sigma_1 \times \ldots \times \sigma_n) \rightarrow \tau}(s_1, \ldots, s_n) : \tau$	if $Z \in \mathcal{M}$, $s_1 : \sigma_1, \ldots, s_n : \sigma_n$, Z has arity n

Moreover, both variables and meta-variables must have a unique type (or type declaration) in s – if both x_σ and x_τ occur in s (with $x \in \mathcal{V} \cup \mathcal{M}$) then $\sigma = \tau$.

A *term* is a meta-term without meta-variables, so its type is found without clause `(meta)`.

The $\lambda x_\sigma.s$ construct *binds* the variable x in s. Note that only variables can be bound, not meta-variables, and there are no binders other than λ. As usual, we work modulo renaming of variables bound by an abstraction operator (α-conversion). The $\cdot$ operator for application associates to the left. We will commonly omit the $\cdot$ and just write $s\ t$ instead of $s \cdot t$. We also omit empty argument lists in meta-variable applications, using Z instead of $Z()$. Write $\mathit{FVar}(s)$ for the set of variables occurring free in s, $\mathit{FMVar}(s)$ for the set of meta-variables occurring in s and $\mathit{FTVar}(s)$ for the set of type variables occurring in s.

A meta-term is a *pattern* if it does not contain any subterms $Z(s_1, \ldots, s_n) \cdot t$ or $(\lambda x.s) \cdot t$, and in subterms $Z(s_1, \ldots, s_n)$ all s_i are distinct bound variables.

Type Substitution A type substitution is a mapping $\theta : \mathcal{A} \rightarrow \mathcal{T}$. For any (meta-)term s let $s\theta$ be s with all type variables α replaced by $\theta(\alpha)$. As an example, $(\mathtt{if}_{\mathtt{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha}\ x_{\mathtt{bool}}\ y_\alpha\ 0_\alpha)\{\alpha \rightarrow \mathtt{nat}\} = \mathtt{if}_{\mathtt{bool} \rightarrow \mathtt{nat} \rightarrow \mathtt{nat} \rightarrow \mathtt{nat}}\ x_{\mathtt{bool}}\ y_{\mathtt{nat}}\ 0_{\mathtt{nat}}$. We say $s \geq t$ if there is a type substitution θ such that $s = t\theta$; thus, we have $\geq$ ("more general than") both on types and meta-terms.

(Meta-)term Substitution A (meta-term) substitution is the homomorphic extension of a mapping $[x^1_{\sigma_1} := s_1, \ldots, x^n_{\sigma_n} := s_n]$, where:

1. each s_i is a term
2. each x^i is either a variable or a meta-variable
3. if x^i is a variable, then σ_i is a type and $s_i : \sigma_i$
4. if x^i is a meta-variable of arity m then σ_i is a type declaration $(\tau_1 \times \ldots \times \tau_m) \longrightarrow \rho$ of arity m and $s_i = \lambda y^1_{\tau_1} \ldots y^m_{\tau_m}.u$ with $u : \rho$.

A substitution replaces the variables and meta-variables in its domain everywhere in the meta-term. When substituting for a meta-variable we additionally perform a development. For example,
$(x \cdot (\lambda y.y))[x := \lambda z.z\ a] = (\lambda z.z\ a)(\lambda y.y)$ whereas $Z(\lambda y.y)[Z := \lambda z.z\ a] = (\lambda y.y)\ a$.
Formally, for any meta-term s the result $s\gamma$ of substituting is generated by the following rules:
$x^i_{\sigma_i}\gamma = s_i$ if x^i is a variable;
$x^i_{\sigma_i}(t_1, \ldots, t_m)\gamma = u[y^1 := t_1, \ldots, y^m := s_m]$ if x_i is a meta-variable and $s_i = \lambda y^1 \ldots y^m.u$;
$f_\sigma\gamma = f_\sigma$ for $f \in \mathcal{F}$;
$(s \cdot t)\gamma = (s\gamma) \cdot (t\gamma)$;
$(\lambda x_\sigma.s)\gamma = \lambda x_\sigma.(s\gamma)$ if no x_τ occurs in domain or range of γ (we can rename x if necessary);
$x_\sigma\gamma = x_\sigma$ if x a variable and x_σ is not one of the $x^i_{\sigma_i}$ (either not $x = x^i$ or not $\sigma = \sigma_i$ for all i);

$Z_\sigma(s_1,\dots,s_n)\gamma = Z(s_1\gamma,\dots,s_n\gamma)$ if Z a meta-variable and Z_σ is not one of the $x^i_{\sigma_i}$.
Note that substitution is well-defined: the step where a meta-term is replaced uses a second substitution, but this second substitution has no meta-variables in its domain.

Context A *context* is a meta-term C with a special symbol $\square_\sigma$ occurring in it. Write $C[s]$ for the meta-term C with $\square_\sigma$ replaced by s; to avoid losing typability s should have type σ.

Eta The relation of restricted η-expansion, $\rightsquigarrow_\eta$, is defined as follows: $C[s] \rightsquigarrow_\eta C[\lambda x_\sigma.s\, x_\sigma]$ if $s : \sigma \rightarrow \tau$ and the following conditions are satisfied:

1. x is a fresh variable;
2. s is neither an abstraction $\lambda x.s'$ nor a meta-variable application $Z(s_1,\dots,s_n)$
3. s in $C[s]$ is not the left part of an application;
4. s in $C[s]$ is not the direct argument of a meta-variable application.

By parts 3 and 4 s is not expanded if it occurs in a sub-meta-term of the form $s\ u$ or $Z_\tau(\dots,s,\dots)$; requirements 2 and 3 guarantee that $\rightsquigarrow_\eta$ always terminates. Therefore every term s has a unique *η-long form* $s{\uparrow_\eta}$ which can be found by applying $\rightsquigarrow_\eta$ until it is no longer possible. We say a term s is in η-long form or has η-long form if $s = s{\uparrow_\eta}$.

Example 2. Some examples: $Z_{(o\times(o\rightarrow o))\longrightarrow a\rightarrow b}(0_o, g_{o\rightarrow o}){\uparrow_\eta} = Z_{(o\times(o\rightarrow o))\longrightarrow a\rightarrow b}(0_o, g_{o\rightarrow o})$.
$(\lambda x.f_{\mathtt{nat}\rightarrow\mathtt{nat}\rightarrow\mathtt{nat}}\ x_{\mathtt{nat}})\ 0\ 0{\uparrow_\eta} = (\lambda xy.f_{\mathtt{nat}\rightarrow\mathtt{nat}\rightarrow\mathtt{nat}}\ x_{\mathtt{nat}}\ y_{\mathtt{nat}})\ 0\ 0$.

Rule Schemes and Rules A *rule scheme* is a pair $l \Rightarrow r$ of meta-terms such that

1. l and r have the same type,
2. all meta-variables and type variables occurring in r also occur in l,
3. l and r are closed,
4. l is a pattern,
5. r has no subterms $(\lambda x.s)\ t$ and
6. l has the form $f\ l_1 \cdots l_n$ with $f \in \mathscr{F}$ and $n \geq 0$.

A rule scheme is a *rule* if it is monomorphic (where a pair of meta-terms is called monomorphic if all types occurring in it are). A set of rule schemes R may yield a set of rules $\mathscr{R}$ in one of two ways. The *standard* set of rules $\mathscr{R}_R$ generated from R is the set of rules $l\theta \Rightarrow r\theta$ with $l \Rightarrow r \in \mathscr{R}$ and θ a type substitution with domain $FTVar(l)$ and only monomorphic types occurring in its range. The *η-long* set of rules $\mathscr{R}_R^\eta$ generated from R is the set of rules $l\ Z^1_{\sigma_1} \cdots Z^n_{\sigma_n}{\uparrow_\eta} \Rightarrow r\ Z^1_{\sigma_1} \cdots Z^n_{\sigma_n}{\uparrow_\eta}$ where $l \Rightarrow r$ is in $\mathscr{R}_R$, $l : \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau$ with τ a sort and $Z^1,\dots,Z^n$ fresh meta-variables. Note that neither applying a type substitution nor η-normalising affects properties (1-6).

Although the rewrite relation is defined using a set of rules rather than rule schemes, it has merit to keep in mind the rule schemes which generated $\mathscr{R}$, since in polymorphic systems sets of rule schemes are often finite while the rules generated from them are not. Since every rule is also a rule scheme, we can always assume a set of rules was generated in one of the above two ways.

Rewrite Relation A set of rules $\mathscr{R}$ generates a *rewrite relation* on monomorphic terms as follows:

`(top)`	$l\gamma \Rightarrow_{\mathscr{R}} r\gamma$	if $l \Rightarrow r \in \mathscr{R}$ and γ a substitution $(**)$
`(app-l)`	$s\,t \Rightarrow_{\mathscr{R}} s'\,t$	if $s \Rightarrow_{\mathscr{R}} s'$
`(app-r)`	$s\,t \Rightarrow_{\mathscr{R}} s\,t'$	if $t \Rightarrow_{\mathscr{R}} t'$
`(lambda)`	$\lambda x.s \Rightarrow_{\mathscr{R}} \lambda x.s'$	if $s \Rightarrow_{\mathscr{R}} s'$
`(beta)`	$(\lambda x.s)\,t \Rightarrow_{\mathscr{R}} s[x := t]$	

(**) Since the rewrite relation is on terms, the domain of γ must contain all meta-variables in l.
The reduction relation is decidable due to the pattern restriction. We say $s \Rightarrow_{\beta} t$ if $s \Rightarrow_{\mathscr{R}} t$ can be derived without clause `(top)` (so the base part is done with `(beta)`). It is well-known that $\Rightarrow_{\beta}$ on itself is terminating and has unique normal forms; write $s{\downarrow_{\beta}}$ for the β-normal form of s.

In addition, we could define a reduction strategy and investigate termination (and other properties) adopting this strategy; common reduction strategies are, for example, innermost ($s \Rightarrow_{\mathscr{R},\text{innermost}} t$ if either $s \Rightarrow_{\mathscr{R},top} t$ and the direct subterms of s cannot $\Rightarrow_{\mathscr{R}}$-reduce, or a direct subterm of s reduces innermost), outermost ($s \Rightarrow_{\mathscr{R},\text{outermost}} t$ if either $s \Rightarrow_{\mathscr{R},top} t$ or s is not a redex and one of its direct subterms reduces outermost) and beta-first ($s \Rightarrow_{\mathscr{R},\text{beta-first}} t$ if either $s \Rightarrow_{\beta} t$ or $s \Rightarrow_{\mathscr{R}} t$ and s is beta-normal).

Requirements (4-6) may be puzzling at first sight, since by placing restrictions on the rules the definition of the system becomes more complicated. However, by disallowing problematic rules like $Z\ 0 \Rightarrow (\lambda x.f(x)) \cdot 1$ it becomes much easier to obtain interesting results, and (as we will see briefly in Section 6.4) we do not truly lose expressivity by placing these limitations.

Example 3. The example system `map` has signature

$$\mathscr{F} = \{\mathtt{nil}_{\mathtt{list}(\alpha)},\ \ \mathtt{cons}_{\alpha\to\mathtt{list}(\alpha)\to\mathtt{list}(\alpha)},\ \ \mathtt{map}_{(\alpha\to\alpha)\to\mathtt{list}(\alpha)\to\mathtt{list}(\alpha)}\}$$

and rules generated in a standard way by the set of rule schemes (taking $\sigma := (\alpha \to \alpha) \to \mathtt{list}(\alpha) \to \mathtt{list}(\alpha)$ and $\tau := \alpha \to \mathtt{list}(\alpha) \to \mathtt{list}(\alpha)$)

$$R = \left\{ \begin{array}{lcl} \mathtt{map}_{\sigma}\ F\ \mathtt{nil}_{\mathtt{list}(\alpha)} & \Rightarrow & \mathtt{nil}_{\mathtt{list}(\alpha)} \\ \mathtt{map}_{\sigma}\ F\ (\mathtt{cons}_{\tau}\ H\ T) & \Rightarrow & \mathtt{cons}_{\tau}\ (F\ H)\ (\mathtt{map}_{\sigma}\ F\ T) \end{array} \right\}$$

We will commonly omit explicit type specification and just assume the most general possible type is used. This way, the second rule becomes: $\mathtt{map}\ F\ (\mathtt{cons}\ H\ T) \Rightarrow \mathtt{cons}\ (F\ H)\ (\mathtt{map}\ F\ T)$.

Extending the signature with symbols $\mathtt{0} : \mathtt{nat}$ and $\mathtt{s} : \mathtt{nat} \to \mathtt{nat}$ we may define a term $\mathtt{map}\ (\lambda x.\mathtt{s}\ x)\ (\mathtt{cons}\ \mathtt{0}\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{nil}))$ (here, `map` has a type $(\mathtt{nat} \to \mathtt{nat}) \to \mathtt{list}(\mathtt{nat}) \to \mathtt{list}(\mathtt{nat})$), which we can reduce as follows:

$$\begin{array}{ll} \mathtt{map}\ (\lambda x.\mathtt{s}\ x)\ \mathtt{cons}\ \mathtt{0}\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{nil})) & \Rightarrow_{\mathscr{R}} \\ \mathtt{cons}\ ((\lambda x.\mathtt{s}\ x)\ \mathtt{0})\ (\mathtt{map}\ (\lambda x.\mathtt{s}\ x)\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{nil})) & \Rightarrow_{\mathscr{R}} \\ \mathtt{cons}\ ((\lambda x.\mathtt{s}\ x)\ \mathtt{0})\ (\mathtt{cons}\ ((\lambda x.\mathtt{s}\ x)\ (\mathtt{s}\ \mathtt{0}))\ (\mathtt{map}\ (\lambda x.\mathtt{s}\ x)\ \mathtt{nil})) & \Rightarrow_{\beta} \\ \mathtt{cons}\ ((\lambda x.\mathtt{s}\ x)\ \mathtt{0})\ (\mathtt{cons}\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0}))\ (\mathtt{map}\ (\lambda x.\mathtt{s}\ x)\ \mathtt{nil})) & \Rightarrow_{\mathscr{R}} \\ \mathtt{cons}\ ((\lambda x.\mathtt{s}\ x)\ \mathtt{0})\ (\mathtt{cons}\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0}))\ \mathtt{nil}) & \Rightarrow_{\beta} \\ \mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ (\mathtt{cons}\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0}))\ \mathtt{nil}) & \end{array}$$

3 Functional Syntax

First-order term rewriting systems are usually written using functional syntax, where every function symbol comes equipped with an arity expressing the number of arguments it takes, for example $add(x,0) \Rightarrow x$.

There are also so-called applicative systems, with a special binary symbol for application and a signature containing function symbols that do not get arguments. The paradigmatic example is Combinatory Logic, with for instance the rewrite rule $Sxyz \Rightarrow (xz)(yz)$. However, t is well-known that going from a functional TRS to its applicative variant, or the other way round, may have consequences for properties such as termination. One of the reasons is that in the applicative variant of a functional system there are more terms. For example, the applicative variant of the example for addition also has *add* 0 as a term, as well as *add* 0 0 0. Neither of these correspond to a term in the functional syntax. See also [3] and [6].

In several higher order formalisms (as we will see in Section 6) terms are functional as well. In our setting this is not the case: all partial applications are allowed. To deal with functional systems we could go two ways: either equip function symbols with a type declaration rather than a type, as is done in [5], or impose an external arity restriction on terms. We have chosen for the second approach because a functional approach complicates the system unnecessarily, especially in view of Theorem 1.

Definition 1 (Arity)**.** Given a set of function symbols $\mathscr{F}$, an arity function ar assigns to each $f_{\sigma_1\to\ldots\to\sigma_n\to\tau} \in \mathscr{F}$ a number $m \leq n$ (where τ is a base type or type variable). A meta-term s *respects* ar if any subterm f_τ occurs in a context $f_\tau\ s_1 \cdots s_n$ in s with $n \geq ar(f)$.

Obviously in a meta-term which respects ar we can write $f(s_1,\ldots,s_n)$ instead of $f\ s_1 \cdots s_n$ without problems. Respecting ar is closed under type substitution.

Theorem 1. *Let ar be an arity function for $\mathscr{F}$ and $\mathscr{R}$ a set of rules over $\mathscr{F}$. If all left- and right-hand sides of $\mathscr{R}$ respect ar then $s \Rightarrow_{\mathscr{R}} t$ implies that t respects ar if s does, and $\Rightarrow_{\mathscr{R}}$ is terminating iff it is terminating on terms which respect ar.*

Note that if rule schemes R respect ar, then so does $\mathscr{R}_R$, and an η-long set of rules respects any arity.

Proof. One direction is obvious. For the other, let *res* be the homomorphic extension of the function $res(f\ s_1 \cdots s_n) = \lambda x_{n+1} \ldots x_{ar(f)}.f\ res(s_1) \cdots res(s_n)\ x_{n+1} \cdots x_{ar(f)}$. For patterns l which respect ar, meta-terms r and substitutions γ we have $res(l\gamma) = l\gamma^{res}$ and $res(r)\gamma^{res} \Rightarrow_\beta^* res(r\gamma)$ (where $\gamma^{res} = \{x \mapsto res(\gamma(x)) | x \in \mathrm{dom}(\gamma)\}$). Therefore $s \Rightarrow_{\mathscr{R}} t$ implies $res(s) \Rightarrow_{\mathscr{R}}^+ res(t)$. □

The requirement that the rules should respect ar is essential: for example, the rule $g\ f_{\mathtt{nat}\to\mathtt{nat}} \Rightarrow g\ f_{\mathtt{nat}\to\mathtt{nat}}$ yields an infinite rewrite sequence, but is terminating when we restrict attention to those terms which respect an arity function $ar(f_{\mathtt{nat}\to\mathtt{nat}}) = 1$. The reason is that $g\ f$ does not match terms of the form $g\ (\lambda x.f\ x)$. Of course, it is not very natural to impose a restriction on the term formation which is not satisfied by the rewrite rules. As long as the left-hand sides of all rewrite rules respect ar we can, however, translate the rules to respect ar (using the *res* function from the proof of Theorem 1) without losing non-termination. Termination is lost in systems with rules like $f_{o\to o}\ Z_o \Rightarrow g_{(o\to o)\to o}\ f_{o\to o}$.

4 Eta-normality

Another common restriction in the literature is to look only at η-long terms. This is convenient for many reasons; for example, if a higher-order term g is just the same as $\lambda x.g\ x$ we only have to consider terms of base type for non-termination. In fact, if the rules are η-long we are free to make this restriction:

Theorem 2. *If $\mathscr{R}$ is η-long, then $\mathscr{R}$ maps η-long terms to η-long terms and $s \Rightarrow_{\mathscr{R}} t$ implies $s \uparrow_\eta \Rightarrow_{\mathscr{R}}^+ \cdot \Rightarrow_\beta^* t \uparrow_\eta$. Therefore $\Rightarrow_{\mathscr{R}}$ is terminating if and only if it is terminating on η-long terms.*

Proof. Writing $\gamma^\uparrow = \{x \mapsto \gamma(x) \uparrow_\eta | x \in \mathrm{dom}(\gamma)\}$ this primarily holds because $l\gamma \uparrow_\eta = l\gamma^\uparrow$ and $r\gamma^\uparrow \Rightarrow_\beta^* r\gamma \uparrow_\eta$ when l, r are η-long meta-terms and l is a pattern; induction on $\Rightarrow_{\mathscr{R}}$ completes the proof. □

However, if $\mathscr{R}$ is not η-long we cannot just transform it to an η-long version without losing out; the counterexample from Section 3, $f_{o\to o}\ Z \Rightarrow g_{(o\to o)\to o}\ f_{o\to o}$, still applies: this system is terminating, but its η-long version is not. On the positive side, however, turning the rules η-long can only lose *termination*, not *non-termination*.

Theorem 3. *Given a set of rule schemes R. If $s \Rightarrow_{\mathscr{R}_R} t$ then $s{\uparrow}_\eta \Rightarrow_{\mathscr{R}_R^\eta} t{\uparrow}_\eta$, so termination of the rewrite relation for $\mathscr{R}_R^\eta$ implies termination of $\mathscr{R}_R$.*

Note that given a set of rules $\mathscr{R}$ we can just take $R := \mathscr{R}$ to apply this Theorem.

Proof. This holds because always $s{\uparrow}_\eta\gamma^{\uparrow} \Rightarrow_\beta^* s\gamma{\uparrow}_\eta$ and when s is a pattern even $s{\uparrow}_\eta\gamma^{\uparrow} = s{\uparrow}_\eta$. □

Thus, we can apply termination techniques on the η-long form of the rules, and if they succeed, we have also proved termination of the original system. However, since there are systems which cannot be handled that way, there is merit in also developing techniques which are not preserved under η-normalisation. Despite this, terms and rules being η-long is an assumption we can usually make without losing too much.

5 Beta-normality

Another pleasant assumption to work with is having no beta-redexes in your terms. Unfortunately, beta-reduction makes an essential difference to termination. Consider for example the system with two rules:

$$\begin{array}{rcl} f\ 0 & \Rightarrow & g\ (\lambda x_{\mathtt{nat}\to\mathtt{nat}}.y_{\mathtt{nat}}.x\ y) \\ g\ (\lambda xy.Z(x,y)) & \Rightarrow & Z(\lambda y_{\mathtt{nat}}.0, f\ 0) \end{array}$$

It is not hard to see that this system is non-terminating. However, if terms are β-normalised after applying each rule, there is no infinite reduction. It gets worse, however. Consider the following system:

$$\begin{array}{rcl} f_1\ x & \Rightarrow & f_2\ x\ x \\ f_2\ a\ x & \Rightarrow & f_3\ x \\ g_1\ (\lambda y.f_3\ Z(y)) & \Rightarrow & Z(\text{hide}\ (\lambda y.f_3\ Z(y))) \\ \text{unhide}\ (\text{hide}\ (\lambda y.Z(y))) & \Rightarrow & g_2\ (\lambda y.Z(y)) \\ g_2\ (\lambda y.f_3\ Z(y)) & \Rightarrow & g_1\ (\lambda y.f_1\ Z(y)) \end{array}$$

This (admittedly highly artificial) system is β-*closed*, that is, it has the property that if s is β-normal and $s \Rightarrow_{\mathscr{R}} t$, then also t is β-normal. Yet non-termination is caused by the possibility of non-β-normal terms. Let $\chi[y] := (\lambda x.a)\ (\text{unhide}\ y)$. Then:

$$\begin{array}{llll} & g_1\ (\lambda y.f_1\ \chi[y]) & \Rightarrow_{\mathscr{R}} & g_1\ (\lambda y.f_2\ \chi[y]\ \chi[y]) \\ \Rightarrow_\beta & g_1\ (\lambda y.f_2\ a\ \chi[y]) & \Rightarrow_{\mathscr{R}} & g_1\ (\lambda y.f_3\ \chi[y]) \\ \Rightarrow_{\mathscr{R}} & \chi[\text{hide}\ (\lambda y.f_3\ \chi[y])] & = & (\lambda x.a)\ (\text{unhide}\ (\text{hide}\ (\lambda y.f_3\ \chi[y]))) \\ \Rightarrow_{\mathscr{R}} & (\lambda x.a)\ (g_2\ (\lambda y.f_3\ \chi[y])) & \Rightarrow_{\mathscr{R}} & (\lambda x.a)\ (\underline{g_1\ (\lambda y.f_1\ \chi[y])}) \end{array}$$

The crux of the example is that due to β-reduction there is a term that reduces to a but also has a subterm unhide y; no counterpart on β-normal terms exists.

When we are only interested in β-normal terms, we can either restrict termination analysis to this subset of terms, or (especially if $\Rightarrow_{\mathscr{R}}$ is not β-closed) use a beta-first reduction strategy.

Theorem 4. *Let $\sqsupset$ be the relation with $s \sqsupset t$ iff s is β-normal, there is t' with $s \Rightarrow_{\mathscr{R}} t'$ and $t = t'\downarrow_\beta$. Then $\sqsupset$ is terminating if and only if $\Rightarrow_{\mathscr{R},\mathrm{beta-first}}$ is.*

Restricting interest to β-normal terms is not just useful for studying termination under a strategy, or for a limited subset of terms. In fact, for any system which satisfies a number of (not too unreasonable) requirements, β-normal termination is all we need. Say a meta-term is *fully extended* if in every meta-application all bound variables occur as arguments (so $\lambda x.f\ Z(x)$ is fully extended whereas $\lambda xy.f\ Z(x)$ is not); a set of rules is considered fully extended if all left-hand sides are. Say a meta-term is *simple functional* if it is β-normal and has no subterms $x\ s$ with $x \in \mathscr{V}$ or $Z(\vec{t})\ s$ with $Z \in \mathscr{M}$; a set of rules is simple functional if both sides of all rules are. A set of rules is second-order if all meta-variables occur with a type $(b_1 \times \ldots \times b_n) \to b_{n+1} \to \ldots \to b_m \to b_{m+1}$ with all b_i data types.

Theorem 5. *Suppose $\mathscr{R}$ is left-linear, fully extended, second-order, η-long and simple functional. For every pair of type constructors $a, b \in \mathscr{B}$ with $ar(a) = n, ar(b) = m$ introduce a new function symbol $P^{a,b} : a(\alpha_1,\ldots,\alpha_n) \to b(\beta_{n+1},\ldots,\beta_{n+m}) \to a(\alpha_1,\ldots,\alpha_n)$ and let $\mathscr{R}^P = \mathscr{R} \cup \{P^{a,b}\ X\ Y \Rightarrow X | a,b \in \mathscr{B}\}$. Then $\mathscr{R}^P$ maps simple functional terms to simple functional terms, and $\mathscr{R}$ is terminating if and only if $\mathscr{R}^P$ is terminating on simple functional terms.*

Proof. In a left-linear, fully extended set of rules we can typically avoid steps inside an application other than functional applications $f\ s_1 \cdots s_n$. Realising that, and only considering η-long terms, we can replace terms $(\lambda x.s)\ (\lambda \vec{y}.t)\ r_1 \cdots r_n$ by $P\ (s[x := \lambda \vec{y}.t]\ \vec{r})\ t$, and reductions go through unhampered. □

Of course, not all systems satisfy the requirements of Theorem 5. However, with some transformations we can in principle assume a beta-first reduction strategy for all (standard) HODRSs:
Let $\mathscr{F}$ be a set of function symbols, R a set of rule schemes and let $@_{(\alpha_1\to\alpha_2)\to\alpha_1\to\alpha_2}$ be a fresh function symbol. Write the rule schemes in R in a functional form, respecting maximal arity. Let $A(s)$ be s with all applications $u\ v$ replaced by $@(u,v)$ – note, here, that function applications $f(s_1,\ldots,s_n)$ are not directly affected. Let $R^A = \{A(l) \Rightarrow A(r) | l \Rightarrow r \in R\} \cup \{@(X,Y) \Rightarrow X\ Y\}$.

Theorem 6. *$\Rightarrow_{\mathscr{R}_R}$ is terminating if and only if $\Rightarrow_{\mathscr{R}_{R^A}}$ is terminating, if and only if $\Rightarrow_{\mathscr{R}_{R^A},\mathrm{beta-first}}$ is terminating.*

Proof. It is not hard to show that always $A(s\gamma) = A(s)\gamma^A$ (where $\gamma^A = \{x \mapsto A(\gamma(x)) | x \in \mathrm{dom}(\gamma)\}$) and therefore $s \Rightarrow_{\mathscr{R}} t$ implies $A(s) \Rightarrow_{\mathscr{R}_{R^A}} \cdot \Rightarrow_\beta^= A(t)$, which is a $\Rightarrow_{\mathscr{R}_{R^A},\mathrm{beta-first}}$ reduction because $A(s)$ is β-normal. On the other hand, if $s \Rightarrow_{\mathscr{R}_{R^A}} t$ then $A^{-1}(s) \Rightarrow_{\mathscr{R}}^* A^{-1}(t)$ (where A^{-1} just replaces all $@(u,v)$ by $u\ v$), with equality only possible if it was an $@(s,t) \Rightarrow_{\mathscr{R}} s\ t$ step). □

In many realistic examples, R^A will actually be the same as R but with a single extra rule scheme – unfortunately a polymorphic rule scheme that generates infinitely many rules. Only when meta-variables have a functional output type, such as in a rule $\mathtt{map}\ F\ (\mathtt{cons}\ H\ T) \Rightarrow \mathtt{cons}\ (F\ H)\ (\mathtt{map}\ F\ T)$, does this transformation really make the system more difficult to use (the right hand side becomes $\mathtt{cons}\ (@\ F\ H)\ (\mathtt{map}\ F\ T)$). There are ways around this, but we do not discuss these here.

6 Transforming Existing Formalisms

We will now discuss the formalisms most commonly used in the context of higher order rewriting, and demonstrate how they can be embedded into the HODRS format and (sometimes) vice versa.

6.1 Contraction Schemes

The first formalism for defining general higher order rewriting was Aczel's *Contraction Schemes* (CSs) [1]. While no typing occurs in the definition, it fits quite well into our own formalism – it is merely restricted to second order, simple functional terms.

Definition *Terms* in a Contraction Scheme are built from an infinite set of variables, and a signature Σ of *forms* $f : [k_1,\dots,k_n]$ (with all $k_i \in \mathbb{N}$), according to the following clauses: (1) a variable is a term, (2) if $f : [k_1,\dots,k_n] \in \Sigma$ and $s_1,\dots,s_n$ are terms, then also $f(\lambda x_{1,1},\dots,x_{1,k_1}.s_1,\dots,\lambda x_{n,1},\dots,x_{n,k_n}.s_n)$ is a term. *Meta-terms* use in addition to variables and forms also meta-variables, each with a fixed arity. Meta-terms are built using the clauses for terms together with (3) if Z is an n-ary meta-variable and $s_1,\dots,s_n$ are meta-terms, then $Z(s_1,\dots,s_n)$ is a meta-term. A *rewrite rule* is a pair $l \Rightarrow r$ of meta-terms such that all meta-variables in r also occur in l. In addition, the left-hand side l of a rule must satisfy the following restrictions: (i) l is closed (that is, all variables occur in the scope of a λ), (ii) l is linear, that is, every meta-variable occurs at most once in l, (iii) l is fully extended, (iv) l has a depth of 1 or 2 (where $Z(\vec{x})$ has depth 0 and $f(\lambda\vec{x_1}.s_1,\dots,\lambda\vec{x_n}.s_n)$ has depth $\max(\{\text{depth}(s_i) \mid 1 \le i \le n\})+1$).

The rewrite relation over a set of rewrite rules R is given by: (1) $f(\lambda\vec{x_1}.s_1,\dots,\lambda\vec{x_i}.s_i,\dots,\lambda\vec{x_n}.s_n) \hookrightarrow_R f(\lambda\vec{x_1}.s_1,\dots,\lambda\vec{x_i}.s_i',\dots,\lambda\vec{x_n}.s_n)$ if $s_i \hookrightarrow_R s_i'$, and (2) $l\gamma \hookrightarrow_R r\gamma$ if $l \Rightarrow r \in R$ and γ is a meta-substitution, that is, a construct $[Z_1 := (\vec{x_1})a_1,\dots,Z_n := (\vec{x_n})a_n]$ containing all meta-variables occurring in l; applying γ on a meta-term s replaces all occurrences of $Z_i(s_1,\dots,s_m)$ by $a_i[x_{i,1} := s_1,\dots,x_{i,m} := s_m]$.

Example 4. We could represent map from Example 3 as the following Contraction Scheme:

$$\begin{array}{rcl} \Sigma & = & \{\mathtt{nil} : [],\ \mathtt{cons} : [0,0],\ \mathtt{map} : [1,0]\} \\ R & = & \left\{ \begin{array}{l} \mathtt{map}(\lambda x.F(x),\mathtt{nil}) \Rightarrow \mathtt{nil}, \\ \mathtt{map}(\lambda x.F(x),\mathtt{cons}(H,T)) \Rightarrow \mathtt{cons}(F(H),\mathtt{map}(\lambda x.F(x),T)) \end{array} \right\} \end{array}$$

Transformation from CS to HODRS We explain how a Contraction Scheme (Σ,R) can be transformed into a HODRS $(\mathscr{F},\mathscr{R})$. Let o be a 0-ary type constructor and write σ_n for the type $o \to \dots \to o \to o$ with in total n arrows (so $n+1$ occurrences of o), and τ_n for the type declaration $(o \times \dots \times o) \longrightarrow o$ (also $n+1$ occurrences of o). A form $f : [k_1,\dots,k_n]$ in Σ is translated into a function symbol $f_{\sigma_{k_1} \to \dots \to \sigma_{k_n} \to o}$ in $\mathscr{F}$. The translation ϕ from CS-style meta-terms to HODRS-style meta-terms is defined as follows:

$$\begin{array}{rcl} \phi(x) & = & x_o (x \in \mathscr{V}) \\ \phi(f(\lambda\vec{x_1}.s_1,\dots,\lambda\vec{x_n}.s_n)) & = & f_{\sigma_{k_1} \to \dots \to \sigma_{k_n} \to o}\ (\lambda\vec{x_1}.\phi(s_1)) \cdots (\lambda\vec{x_n}.\phi(s_n)) \\ \phi(Z(s_1,\dots,s_n)) & = & Z_{\tau_n}(\phi(s_1),\dots,\phi(s_n)) \end{array}$$

For the translation of rewrite rules, we first observe that in a CS the right-hand side may contain free variables. To solve this we add a fresh constant v with arity [] to Σ and replace all free variables in all right-hand sides by v. Since v does not occur in any left-hand side and contraction schemes are left-linear this has no significant influence on the rewrite relation (and does not affect termination).

Having done this, we can simply take $\mathscr{R} = \{\phi(l) \Rightarrow \phi(r) | l \Rightarrow r \in R\}$. The resulting HODRS is a left-linear, fully extended, second-order, η-long, and simple functional HODRS with only one base type.

The resulting reduction relation $\Rightarrow_{\mathscr{R}}$ reduces simple functional terms to simple functional terms again, and $\hookrightarrow_R$ is terminating if and only if $\Rightarrow_{\mathscr{R}}$ is terminating on simple functional terms, which is certainly the case if it is terminating on β-normal terms, if and only if $\Rightarrow_{\mathscr{R},\text{beta-first}}$ is terminating. If Σ contains at least one symbol which takes two or more arguments, this is even an equivalence.

Transformation from HODRS to CS Following Theorem 5 a given left-linear, fully extended second order η-long and simple functional HODRS which satisfies requirement (iv) and has only one base type is terminating if and only if the corresponding CS with a single additional rule $P(X,Y) \Rightarrow X$ is terminating. Termination results can be transferred to systems which under collapsing of sorts into the single sort o satisfy these requirements (but non-termination results cannot, as collapsing might lose termination).

6.2 Combinatory Reduction Systems

Contraction Schemes were generalised in 1980 by Klop to *Combinatory Reduction Systems* [8]. These CRSs, further investigated in [9], are still popular today. However, as CRSs drop restrictions on terms as well as on rules, the terms admitted in a CRS will in general not be typable. This untyped nature makes them less interesting from a termination point of view (as most usual examples of CRSs will not be terminating). Nevertheless, there is an embedding into the HODRS formalism.

Definition In a CRS, a term is built from an infinite set of variables and a signature Σ of function symbols, each with a fixed arity $n \in \mathbb{N}$, by the following rules: (1) all variables are terms, (2) if s is a term, then so is $\lambda x.s$, (3) if $f : n \in \Sigma$ and $s_1,\dots,s_n$ are terms, then $f(s_1,\dots,s_n)$ is a term. Meta-terms are defined by these three clauses and additionally: (4) if Z is a meta-variable of arity n, and $s_1,\dots,s_n$ are meta-terms, then $Z(s_1,\dots,s_n)$ is a meta-term. Rules are pairs $l \Rightarrow r$ of closed meta-terms, where in subterms $Z(s_1,\dots,s_n)$ all s_i are bound variables. The rewrite relation is given by: $l\gamma \hookrightarrow_R r\gamma$ if $l \Rightarrow r \in R$ and γ a substitution whose domain consists of the meta-variables in l, $\lambda x.s \hookrightarrow_R \lambda x.s'$ if $s \hookrightarrow_R s'$ and $f(s_1,\dots,s_i,\dots,s_n) \hookrightarrow_R f(s_1,\dots,s_i',\dots,s_n)$ if $s_i \hookrightarrow_R s_i'$. The primary difference with the schemes from Section 6.1 is the admission of abstractions as valid terms, and since terms are untyped, it is this which causes difficulty.

Example 5. We could represent map as the system with

$$\begin{array}{rcl} \Sigma & = & \{\mathtt{nil}:0,\ \mathtt{cons}:2,\ \mathtt{map}:2\} \\ R & = & \left\{ \begin{array}{l} \mathtt{map}(\lambda x.F(x),\mathtt{nil}) \Rightarrow \mathtt{nil}, \\ \mathtt{map}(\lambda x.F(x),\mathtt{cons}(H,T)) \Rightarrow \mathtt{cons}(F(H),\mathtt{map}(\lambda x.F(x),T)) \end{array} \right\} \end{array}$$

This system is non-terminating: abbreviating $\omega := \lambda x.\mathtt{map}(x,\mathtt{cons}(x,\mathtt{nil}))$ and $\Omega := \mathtt{map}(\omega,\mathtt{cons}(\omega,\mathtt{nil}))$ we have $\Omega \hookrightarrow_R \mathtt{cons}(\Omega,\mathtt{map}(\omega,\mathtt{nil}))$.

Transformation from CRS to HODRS We cannot transform a term in the obvious way, because both $f(0)$ and $f(\lambda x.0)$ are valid terms – not both can be typable. Therefore we introduce a special symbol T to "flatten" abstractions. Let $\mathscr{F} = \{\mathtt{T}_{(o\to o)\to o}\} \cup \{f_{\sigma_n} | f : n \in \Sigma\}$ (here σ_n is again the type $o \to \dots \to o \to o$ with in total $n+1$ occurrences of o). Let ϕ be the function mapping CRS-style (meta-)terms to our (meta-)terms as follows:

$$\begin{array}{lcl} \phi(x) & = & x_o\ (x \in \mathscr{V}) \\ \phi(f(s_1,\dots,s_n)) & = & f\ \phi(s_1)\cdots\phi(s_n) \\ \phi(\lambda x.s) & = & \mathtt{T}\ (\lambda x.\phi(s)) \\ \phi(Z(s_1,\dots,s_n)) & = & Z_{\tau_n}(\phi(s_1),\dots,\phi(s_n)) \end{array}$$

Defining $\mathscr{R} := \{\phi(l) \Rightarrow \phi(r) | l \Rightarrow r \in R\}$ it is easily seen that $\Rightarrow_{\mathscr{R}}$ is β-closed and $\hookrightarrow_R$ is terminating if $\Rightarrow_{\mathscr{R}}$ is terminating on β-normal terms; if Σ has any symbol of arity 2 or more this is an equivalence.

Note In the original definition of CRSs [8] function symbols did not have an arity; instead, terms used an applicative format. This is not comparable to the version of application used in HODRSs: by the lack of typing terms could be f, $f\,x_1$, $f\,x_1\,x_2,\dots$ We can deal with this by introducing a symbol `apply` of arity 2 which encodes application (so these terms become f, $\mathtt{apply}(f,x)$ and $\mathtt{apply}(\mathtt{apply}(f,x_1),x_2),\dots$).

6.3 Pattern Higher-order Rewrite Systems

Higher Order Rewrite Systems, which are rewrite relations on typed terms modulo β-and η-equality, were first introduced in [13]. Following Wolfram [18] the restrictions on the rules were dropped in [12] for studying further properties. However, the unrestricted system admits many rules which are hard to reason about, as in general the rewrite relation will not be decidable; contrary to our aims, any formalism which includes HRSs in their totality would be hard to obtain immediate results on, other than on a very abstract level. Therefore we instead look at *pattern HRSs*, which pose a restriction on the rules that corresponds with the restriction in the original definition of HRSs. Pattern HRSs (PRSs) form a very common and natural class, with many interesting results of their own (such as those discussed in [12]).

Definition A PRS is defined as a rewrite relation on higher order terms in long β/η-normal form, that is, terms in η-long form which do not have subterms of the form $(\lambda x.s)\cdot t$; every term s has a unique long β/η-normal form $s\updownarrow_\beta^\eta$. A rewrite rule is a pair $l \Rightarrow r$ of (monomorphic) base-type terms, such that $FVar(r) \subseteq FVar(l)$ and l is a pattern; that is, every free occurrence of a variable F in l is in a base type subterm $F\ x_1\cdots x_n$ with all x_i distinct bound variables. The rules generate the relation $\hookrightarrow_{\mathscr{R}}$ as follows: $s \hookrightarrow_{\mathscr{R}} t\updownarrow_\beta^\eta$ if $s \overline{\hookrightarrow}_{\mathscr{R}} t$ can be derived with clauses (app-l), (app-r), (lambda) and additionally:

(top-hrs) $l\gamma\updownarrow_\beta^\eta \overline{\hookrightarrow}_{\mathscr{R}} r\gamma\updownarrow_\beta^\eta$ for $l \Rightarrow r \in \mathscr{R}$ and γ a substitution.

Example 6. The standard system `map` can be implemented as follows:

$$\begin{array}{rcl} \Sigma & = & \{\mathtt{nil}:\mathtt{list},\ \mathtt{cons}:\mathtt{nat}\to\mathtt{list}\to\mathtt{list},\ \mathtt{map}:(\mathtt{nat}\to\mathtt{nat})\to\mathtt{list}\to\mathtt{list}\} \\ R & = & \left\{ \begin{array}{l} \mathtt{map}\ (\lambda x.F\ x)\ \mathtt{nil} \Rightarrow \mathtt{nil}, \\ \mathtt{map}\ (\lambda x.F\ x)\ (\mathtt{cons}\ H\ T) \Rightarrow \mathtt{cons}\ (F\ H)\ (\mathtt{map}\ (\lambda x.Fx)\ T) \end{array} \right\} \end{array}$$

Note that we have assumed a fixed type `nat` in place of the type variable α in Example 3; to deal with such polymorphism we would have to introduce infinitely many similar rules.

Transformation from HRS to HODRS The rules transform naturally to our format; just replace subterms $F\ s_1\cdots s_n$ with F a free variable in either side of the rules by $F'(s_1,\dots,s_n)$ (where F' is a meta-variable typed correspondingly). The requirements on the left-hand sides of the rules are satisfied by the pattern restriction. Thus we define $\mathscr{R}$ in a natural way. Although HRS-terms are more restrictive than HODRS-terms, Theorem 2 allows us to assume terms are η-long. β-normality can be handled with a reduction strategy as discussed in Section 5. We see: $\hookrightarrow_R$ is terminating if and only if $\Rightarrow_{\mathscr{R},\text{beta-first}}$ is.

Transformation from HODRS to HRS Termination of an η-long HODRS using a beta-first reduction strategy is equivalent with termination of the corresponding HRS (where meta-variable applications are expanded, so for example `map` F `nil` becomes `map` $(\lambda x.F(x))$ `nil`). Note also Theorems 3 and 6 which can be used to transform a system without strategy into an HRS.

6.4 Algebraic Functional Systems

Jouannaud's and Okada's Algebraic Functional Systems, as defined in [4], are in syntax quite close to functional programs in various languages. AFSs are polymorphic systems with β-reduction as a separate step. There are several variations of the original format in the literature, especially with respect to the form of polymorphism used. We follow the definition in [5], which is not restrictive in the terms it allows and uses the same definition of polymorphism as we do.

Definition Terms in an AFS are defined in a functional way: instead of a type, function symbols f come equipped with a type declaration. Terms are built with clauses `(var)`, `(abs)`, `(app)` and the altered clause

$$(\mathtt{func})\ f_{(\sigma_1 \times \ldots \times \sigma_n) \longrightarrow \tau}(s_1, \ldots, s_n) : \tau \text{ if } f_\rho \in \mathscr{F} \text{ and } \rho \geq (\sigma_1 \times \ldots \times \sigma_n) \longrightarrow \tau$$

A *rule* is simply a pair of (polymorphic) terms $l \Rightarrow r$ such that l and r have the same type and all variables and type variables occurring in r also occur in l; $\hookrightarrow_R$ is the smallest monomorphic relation such that $l\theta\gamma \hookrightarrow_R r\theta\gamma$ for all $l \Rightarrow r \in R$, type substitutions θ and substitutions γ, and $(\lambda x.s)\ t \hookrightarrow_R s[x := t]$.

Example 7. The standard `map` example is much like Example 3, with

$$\Sigma = \{\mathtt{nil}_{\mathtt{list}(\alpha)},\ \mathtt{cons}_{\alpha \to \mathtt{list}(\alpha) \to \mathtt{list}(\alpha)},\ \mathtt{map}_{(\alpha \to \alpha) \to \mathtt{list}(\alpha) \to \mathtt{list}(\alpha)}\}$$

and set of rules (taking $\sigma := (\alpha \to \alpha) \to \mathtt{list}(\alpha) \to \mathtt{list}(\alpha)$ and $\tau := \alpha \to \mathtt{list}(\alpha) \to \mathtt{list}(\alpha)$):

$$R = \left\{ \begin{array}{l} \mathtt{map}_\sigma\ F\ \mathtt{nil}_{\mathtt{list}(\alpha)} \Rightarrow \mathtt{nil}_{\mathtt{list}(\alpha)}, \\ \mathtt{map}_\sigma\ F\ (\mathtt{cons}_\tau\ H\ T) \Rightarrow \mathtt{cons}_\tau\ (F\ H)\ (\mathtt{map}_\sigma\ F\ T) \end{array} \right\}$$

Transformation from AFS to HODRS For those AFSs where the rules are in β-normal form and do not have head-variables (which is the case for the vast majority of examples used in the literature on AFSs, for example in those systems originating from functional programs), we can simply replace the variables in the rules by meta-variables of the same type, and replace $f(s_1, \ldots, s_n)$ by $f\ s_1 \cdots s_n$; this gives a set of rule schemes R', and by Theorem 1 the HODRS with rules $\mathscr{R}_{R'}$ is terminating if and only if the original AFS is. If the rules do not satisfy these restrictions, we modify them first, by making application explicit (comparable to what was done in Theorem 6). We use the following transformation:

- Let $S = \{\sigma | \exists l \Rightarrow r \in R[l$ has a subterm $x_\sigma\ u$ with $x \in FVar(l)$ or either l or r has a subterm $(\lambda x.u)\ v$ and $\lambda x.u : \sigma]\}$.
- For every rule with a subterm $u\ v$ with $u : \tau$ such that τ unifies with some type $\sigma \in S$, but not $\sigma \geq \tau$, add a rule $l\theta \Rightarrow r\theta$, where θ is the smallest type substitution that unifies τ with σ. Continue doing this until no new rules are added (this process is finite if S is, otherwise it has a limit).
- For $\sigma \to \tau \in S$, introduce new function symbols $@^{\sigma,\tau}_{(\sigma \to \tau \times \sigma) \longrightarrow \tau}$.
- In all rules $l \Rightarrow r$, replace in either side occurrences $u\ v$ by $@^{\sigma,\tau}(u, v)$ if $u : \rho$ and $S \ni \sigma \to \tau \geq \rho$. In addition, add rules $@^{\sigma,\tau}(X, Y) \Rightarrow X\ Y$ for all $\sigma \to \tau \in S$.

This means we replace applications by an explicit function symbol, but only for those types where it is necessary. Note that in a monomorphic AFS, the second step can be skipped.

Transformation from HODRS to AFS Every HODRS with rules $\mathscr{R}_R$ (generated from rule schemes R) where all meta-variables have arity 0 (*parameter-free HODRS*) is the translation of an AFS (usually of more than one), and thus its termination can be analysed with AFS-techniques. Typically, we would analyse the AFS corresponding to a maximal arity function.

Remarks HODRSs originating from HRSs or contraction schemes will usually not be parameter-free; for example, the HRS for `map` leads to a rule `map` $(\lambda x.F(x))$ `nil`. However, as long as meta-variables only occur in the form $\lambda\vec{x}.Z(\vec{x})$ the system can be handled by the following transformation:

(Transformation) *For every meta-variable $Z_{(\sigma_1\times\ldots\times\sigma_n)\longrightarrow\tau}$ occurring in a set of rule schemes R, let $Z'_{\sigma_1\to\ldots\to\sigma_n\to\tau}$ be a meta-variable of arity 0. In every rule scheme, replace $\lambda\vec{x}.Z(\vec{x})$ on the left by Z' and replace $Z(s_1,\ldots,s_n)$ on the right by $Z'\ s_1\cdots s_n$.*

Theorem 7. *Let R be a set of rule schemes such that meta-variables in the left hand sides only occur in the form $\lambda\vec{x}.Z(\vec{x})$, and R' the system transformed as above. If $s \Rightarrow_{\mathscr{R}_R} t$ then $s \Rightarrow_{\mathscr{R}_{R'}} \cdot \Rightarrow_\beta^* t$, so if $\Rightarrow_{\mathscr{R}_{R'}}$ is terminating then so is $\Rightarrow_{\mathscr{R}_R}$; moreover, $\Rightarrow_{\mathscr{R}_R,\mathrm{beta-first}}$ is terminating if and only if $\Rightarrow_{\mathscr{R}_{R'},\mathrm{beta-first}}$ is.*

6.5 Applicative Systems

Another style of higher order rewriting systems is *applicative* rewriting. In this style there are usually types and functional variables, but no abstractions or other kinds of variable binders. We discuss some of the many variations of applicative rewriting here.

STTRS *Yamada's Simply Typed Term Rewriting Systems* [19] use a version of typing and application rather different from ours: a type can either be a fixed *base type* or has the form $(\sigma_1 \times \ldots \times \sigma_n) \to \sigma_0$ with $n > 0$, $\sigma_0,\ldots,\sigma_n$ types. Thus, we may have types like $(\sigma_1 \times \sigma_2) \to (\tau_1 \times \tau_2) \to \rho$. Given a set Σ of typed constants and a set $\mathscr{V}$ of typed variables, terms are expressions typable by the following rules: (1) $a : \sigma$ if $a_\sigma \in \Sigma \cup \mathscr{V}$, (2) if $s_0 : (\sigma_1 \times \ldots \times \sigma_n) \to \sigma_0$, and $s_1 : \sigma_1,\ldots,s_n : \sigma_n$, then $(s_0 \cdot s_1 \cdots s_n) : \sigma_0$, then $(s_0 \cdot s_1 \cdots s_n) : \sigma_0$ $(n > 0)$. A rule is just a pair of terms $l \Rightarrow r \in R$ of the same type, such that all variables in r also occur in l, and l has the form $(f \cdot s_1 \cdots s_n)$ with $f_{(\sigma_1\times\ldots\times\sigma_n)\to\sigma_0} \in \Sigma$. The rewrite relation $\hookrightarrow_R$ is the smallest monotonic relation such that $l\gamma \hookrightarrow_R r\gamma$ for all $l \Rightarrow r \in R$ and substitutions γ.

Example 8. `map` could be implemented as follows:

$$\begin{array}{rcl} \Sigma & = & \{\mathtt{nil} : \mathtt{list},\ \mathtt{cons} : (\mathtt{nat} \times \mathtt{list}) \to \mathtt{list},\ \mathtt{map} : ((\mathtt{nat} \to \mathtt{nat}) \times \mathtt{list}) \to \mathtt{list}\} \\ R & = & \left\{ \begin{array}{l} \mathtt{map}\ F\ \mathtt{nil} \Rightarrow \mathtt{nil}, \\ \mathtt{map}\ F\ (\mathtt{cons}\ H\ T) \Rightarrow \mathtt{cons}\ (F\ H)\ (\mathtt{map}\ F\ T) \end{array} \right\} \end{array}$$

Note that the F in either `map` rule cannot be instantiated by an abstraction, however, as abstractions are not present in STTRSs.

Example 9. The system with $\Sigma = \{g : (o \times o) \to o \to o\}$ and $R = \{g\ X \Rightarrow X\}$ is terminating as an STTRS, even though the corresponding system as a HODRS leads to non-termination.

TRS$_{\mathrm{hv}}$ *Kusakari's Term Rewriting Systems with Higher-Order Variables* [10] are untyped systems. Terms are built from a countably enumerable set of variables V and a finite set of function symbols Σ, according to the rule that $a(s_1,\ldots,s_n)$ is a term if $a \in V \cup \Sigma$ and $s_1,\ldots,s_n$ are terms. Applying a substitution γ with $\gamma(a) = b(t_1,\ldots,t_m)$ for a in $a(s_1,\ldots,s_n)$ gives $b(t_1,\ldots,t_m,s_1\gamma,\ldots,s_n\gamma)$, so this behaves like the application we are used to. Reductions can be done at the top of a term, or in some of the s_i in $a(s_1,\ldots,s_n)$. It is not possible to directly reduce $a(s_1,\ldots,s_k)$ in $a(s_1,\ldots,s_n)$ (with $k < n$).

Example 10. `map` in a TRS_{hv} would be implemented as follows:

$$\begin{array}{rcl} \Sigma & = & \{\mathtt{map},\ \mathtt{cons},\ \mathtt{nil}\} \\ R & = & \left\{ \begin{array}{l} \mathtt{map}(F,\mathtt{nil}) \Rightarrow \mathtt{nil}, \\ \mathtt{map}(F,\mathtt{cons}(H,T)) \Rightarrow \mathtt{cons}(F(H),\mathtt{map}(F,T)) \end{array} \right\} \end{array}$$

STRS *Kusakari's Simply-typed Term Rewriting Systems* extend TRS_{hv}s by attaching a simple type to all function symbols (where a simple type is either a base type or $\sigma \to \tau$ with both σ and τ simple types) and requiring terms to be typable by the usual requirements.

Later extensions also include a product type $\sigma_1 \times \ldots \times \sigma_n$, and a tuple symbol. In our terminology, STRSs come equipped with an enumerable number of type constructors $prod_n$ (of arity n, for $n \in \mathbb{N}$), and an enumerable number of tuple symbols $\mathtt{tu}_n : prod_n(\alpha_1, \ldots, \alpha_n)$ with $\alpha_1, \ldots, \alpha_n$ type variables.

Remarks We have presented some of these applicative systems, but will not provide an embedding into the HODRS formalism. While it is possible to define a translation (and after removing head variables in the right-hand sides the transformation would likely not be too complicated), there is arguably little merit in doing so. Because the HODRS format is significantly more free than the ones in this section (mostly due to the presence of β-reduction), it is likely that far stronger results can be obtained by studying applicative systems directly, or otherwise their embedding into a general applicative system or even into first-order term rewriting.

7 A Higher Order Recursive Path Ordering

The *recursive path ordering*, a common syntactic termination method, has been extended to AFSs in a long line of research, starting with HORPO [5] and culminating in CPO [2]. Any of these definitions can be extended to HODRSs; we consider as an example HORPO, the first version. Given a well-founded ordering on function symbols, HORPO defines a well-founded ordering on terms with polymorphic simple types.

It is important to realise that HORPO is nothing more or less than an ordering on terms – and terms in the format of [5] are also terms in our formalism, if we write them using applicative notation. Therefore we can use the result as it stands, without any need to redo the well-foundedness proof.

Formally, we need a minor transformation, as the formalism presented in this paper is more free in the types it allows. For any type σ, let $collapse(\sigma)$ be the type σ with all subtypes of the form $b(\sigma_1, \ldots, \sigma_n)$ replaced by the single data type o (for example, $collapse(a \to b(\alpha) \to \alpha) = o \to o \to \alpha$ if α is a type variable). Given a set of rule schemes R respecting the arity function ar, let $\phi(s)$ be s with all types occurring in it collapsed, and written in functional notation.

Theorem 8. *Let $\mathscr{R} = \mathscr{R}_R$ be a set of rules generated from R. Then $\Rightarrow_{\mathscr{R}}$ is terminating if $\phi(l) >^* \phi(r)$ can be proved for all rule schemes $l \Rightarrow r \in R$ with the rules of HORPO and in addition: $s \geq t$ if one of the following holds: (1) $s > t$, (2) $s = t$, (3) $s = Z(s_1, \ldots, s_n)$ and $t = Z(t_1, \ldots, t_n)$ and all $s_i \geq t_i$, (4) $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$ and all $s_i \geq t_i$, (5) $s = s_1\ s_2$, $t = t_1\ t_2$ and each $s_i \geq t_i$, (6) $s = \lambda x.u$, $t = \lambda x.v$ and $u \geq v$.*

Proof. It is easy to see with induction over the definition of HORPO that this extension corresponds with the original definition (which only has rules (1) and (2) for $\geq$) when comparing terms. It is also not hard to derive that thes extension is closed under both type and term substitution. Therefore, writing

$\succ$ for the extended version of the HORPO relation, $\phi(l) \succ^+ \phi(r)$ implies $\phi(l\theta\gamma) = \phi(l)\theta^{collapse}\gamma^{\phi} > \phi(r)\theta^{collapse}\gamma^{\phi} = \phi(r\theta\gamma)$. Therefore any infinite $\mathscr{R}_R$ reduction leads to an infinite decreasing $>_{HORPO}$ reduction, contradicting well-foundedness of the latter. $\square$

Later versions of HORPO introduce a type ordering and a stronger relation on the terms. Unfortunately, the last version (CPO, as defined in [2]) uses a variation of typing where function symbols are required to have a data type as output type. Therefore we can not use this relation directly to order polymorphic rule schemes (at least, not when a function symbol in any rule has a type variable as output type, such as a rule $\mathtt{if}_{\mathtt{bool}\to\alpha\to\alpha}\ \mathtt{true}\ X\ Y \Rightarrow X$). Interpreting every monomorphic data type as a different "sort symbol", we can easily obtain the following result:

Theorem 9 (CPO on meta-terms)**.** *Let $(>, \geq)$ be the primary type-respecting ordering relations defined by CPO, and replace $\geq$ by the relation defined by the following clauses:*

1. *$s : \sigma \geq t : \tau$ if $\sigma \geq_T \tau$ and $s > t$*
2. *$s : \sigma \geq t : \tau$ if $\sigma =_T \tau$ and one of the following holds:*
   (a) $s = \lambda x.s'$, $t = \lambda x.t'$ and $s' \geq t'$;
   (b) $s = f\ s_1 \cdots s_n$, $t = f\ t_1 \cdots t_n$ and $s_1 \geq t_1, \ldots, s_n \geq t_n$ (with $f \in \mathscr{V} \cup \mathscr{F}$);
   (c) $s = Z_\sigma(s_1, \ldots, s_n)$, $t = Z_\sigma(t_1, \ldots, t_n)$ and $s_1 \geq t_1, \ldots, s_n \geq t_n$.

If $l{\uparrow_\eta} >^+ r{\uparrow_\eta}$ for all rules $l \Rightarrow r \in \mathscr{R}$, then $\Rightarrow_{\mathscr{R}}$ is a terminating relation.

Given a set of rule schemes generating $\mathscr{R}$ it is not immediately clear how to (automatically) prove that $l{\uparrow_\eta} > r{\uparrow_\eta}$ always holds, since the CPO ordering itself is not defined on polymorphic terms and not preserved under eta-normalising. One way to use Theorem 9 when presented with finite rule schemes is to define a new relation $\succ$ on polymorphic meta-terms (for example using a subset or alteration of the CPO clauses) and demonstrate that $l \succ r$ implies $l\theta{\uparrow_\eta} > r\theta{\uparrow_\eta}$ for all type substitutions θ. However, such a definition is beyond the scope of this paper.

8 Concluding Remarks

In this paper we have presented a new formalism for higher order rewriting and embedded the most common ways of higher order rewriting into it. We have additionally provided a number of theorems to make it easier to reason about the formalism, and demonstrated that the system is not too general to obtain real results by extending the computability path ordering to it.

Having derived a result in one of the many higher-order systems, it should be relatively easy to see which aspects of the formalism are indispensable in the proof, and thus how to extend the result to other formalisms.

References

[1] P. Aczel (1978): *A General Church-Rosser Theorem*. University of Manchester.

[2] F. Blanqui, J.-P. Jouannaud & A. Rubio (2008): *The Computability Path Ordering: The End of a Quest*. In: *Lecture Notes in Computer Science (CSL '08)*, pp. 1–14.

[3] Nao Hirokawa, Aart Middeldorp & Harald Zankl (2008): *Uncurrying for Termination*. In: *Proceedings of the 15th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning*, *Lecture Notes in Artificial Intelligence* 5330, Springer-Verlag, Doha, pp. 667–681.

[4] J.-P. Jouannaud & M. Okada (1991): *A computation model for executable higher-order algebraic specification languages*. In: *Proceedings of the 6th annual IEEE Symposium on Logic in Computer Science (LICS'91)*, IEEE Computer Society Press, Amsterdam, The Netherlands, pp. 350–361.

[5] J.-P. Jouannaud & A. Rubio (1999): *The higher-order recursive path ordering*. In: *Proceedings of the 14th annual IEEE Symposium on Logic in Computer Science (LICS '99)*, Trento, Italy, pp. 402–411.

[6] R. Kennaway, J.W. Klop, M.R. Sleep & F.J de Vries (1996): *Comparing curried and uncurried rewriting*. *Journal of Symbolic Computation* 21(1), pp. 15–39.

[7] Z. Khasidashvili (1990): *Expression Reduction Systems*. In: *Proceedings of I. Vekua Institute of Applied Mathematics*, 36, Tblisi, Georgia, pp. 200–220.

[8] J.W. Klop (1980): *Combinatory Reduction Systems*, *Mathematical Centre Tracts* 127. CWI, Amsterdam, The Netherlands. PhD Thesis.

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

[10] K. Kusakari (2001): *On proving termination of term rewriting systems with higher-order variables*. *IPSJ Transactions on Programming* 42(SIG 7 PRO11), pp. 35–45.

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

[12] R. Mayr & T. Nipkow (1998): *Higher-Order Rewrite Systems and their Confluence*. *Theoretical Computer Science* 192, pp. 3–29.

[13] T. Nipkow (1991): *Higher-order critical pairs*. In: *Proceedings of the 6th annual IEEE Symposium on Logic in Computer Science (LICS '91)*, Amsterdam, The Netherlands, pp. 342–349.

[14] V. van Oostrom (1994): *Confluence for abstract and higher-order rewriting*. Ph.D. thesis, Vrije Universiteit, Amsterdam, The Netherlands.

[15] F. van Raamsdonk (2001): *On Termination of Higher-Order Rewriting*. In: A. Middeldorp, editor: *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA 2001)*, *LNCS* 2051, Utrecht, The Netherlands, pp. 261–275.

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

[17] Wiki: *Termination Portal*. `http://www.termination-portal.org/`.

[18] D. Wolfram (1993): *The Clausal Theory of Types*, *Cambridge Tracts in Theoretical Computer Science* 21. Cambridge University Press, Cambridge, United Kingdom.

[19] T. Yamada (2001): *Confluence and termination of simply typed term rewriting systems*. In: A. Middeldorp, editor: *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA '01)*, *LNCS* 2051, Springer Verlag, Utrecht, The Netherlands, pp. 338–352.