A Recursive Path Ordering for Higher-Order Terms in η-Long β-Normal Form*

Jean-Pierre Jouannaud†** and **Albert Rubio**‡

† LRI, Bat. 490, CNRS/Université de Paris Sud, 91405 Orsay, FRANCE
Email: Jean-Pierre.Jouannaud@lri.fr
‡ Technical University of Catalonia, Pau Gargallo 5, 08028 Barcelona, SPAIN
Email: rubio@lsi.upc.es

Abstract: This paper extends the termination proof techniques based on rewrite orderings to a higher-order setting, by defining a recursive path ordering for simply typed higher-order terms in η-long β-normal form. This ordering is powerful enough to show termination of several complex examples.

1 Introduction

Higher-order rewrite rules are used in various programming languages and logical systems. In functional languages like Elf, or theorem provers like Isabelle, higher-order rewrite rules define functions by higher-order pattern matching. Higher-order rewriting in this sense enjoys a theory which parallels the usual theory of first-order rewriting. In particular, the main property of first-order rewriting, the critical pair lemma, still holds for some restricted cases [12].

Computer systems like RRL [7], *Saturate* [13] or CiME [11] make available semi-automated techniques for proving termination of first-order rewrite rules by comparing their left and right hand sides in some reduction ordering, of which the most popular one is the recursive path ordering [4]. Its principle is to generate recursively an ordering on terms from a user-defined ordering on function symbols called a precedence.

Our goal is to develop similar techniques for the higher-order case, thus further closing the gap between the practical needs and the existing results [9, 10, 3]. These orderings will of course need to be compatible with the underlying typed λ-calculus. Since higher-order pattern matching is used, compatibility with $\beta\eta$-conversion implies that the ordering operates on $\beta\eta$-equivalence classes of terms. This can be achieved by defining an ordering on canonical representatives, like terms in β-normal η-long form.

Our contribution is precisely the definition of a recursive path ordering for higher-order terms operating on terms in β-normal η-long form. This ordering extends, on one hand a precedence on the function symbols, on the other hand a

* This work was partly supported by the ESPRIT Basic Research Action CCL and the EU Human Capital and Mobility Network *Console.*
** This work results from a stay of the author at the Technical University of Catalonia.

well-founded ordering on the type structure compatible with the signature, that is, such that the output type of any function symbol should be bigger than or equal to its arguments type. Terms are compared by type first, then by head function symbol, before the comparison can proceed recursively. In practice, the ordering on types can itself be a recursive path ordering generated by an arbitrary precedence between the basic types and the arrow, or by interpreting a type expression by its maximal basic type. Several examples are carried out.

This idea of extending a recursive path ordering to higher-order terms in η-long β-normal form was first explored in [9]. The authors restricted their study to patterns in the sense of Miller, and this restriction survived in subsequent work [8, 10]. We were able to get rid of this superflous assumption by proving the properties of our ordering for ground terms. For instance, we prove that (a first approximation of) our ordering enjoys the subterm property for ground terms. This does not contradict the fact that a term $X(a)$ is not greater than a under our ordering on terms in η-long β-normal form. The point is that the term $X(a)$ has ground instances whose η-long β-normal form is not a superterm of a. A similar phenomenon occurs regarding the monotonicity property. Finally these properties on ground terms together with stability under ground substitutions and well-foundedness for ground terms ensure the correctness of the ordering as a termination proof method.

Our ordering is no panacea, unfortunately. There are many important examples such as the apply function or Gödels recursors that cannot be oriented with our current definition. A careful analysis points at a potential remedy, an original, powerful notion of higher-order subterm discussed in [8]. We believe that its use in our definition would overcome the limitations of our current proposal.

Our framework is described in section 2 and the ordering in section 3. Section 4 is devoted to compatible orderings on types and examples of application. A comparison with previous work is given in section 5 and conclusions and future work in section 6.

We expect the reader to be familiar with the basic concepts and notations of term rewriting systems [5] and typed lambda calculi [1, 2].

2 Algebraic λ-Terms

To introduce the language we are going to investigate, we define successively types, terms, typing rules, and computation rules. For simplicity, we consider only one type operator, namely $\rightarrow$, although our results accommodate other type operators as well, e.g., product types and sum types.

2.1 Types

We are first given a set $\mathcal{S}$ of *sorts*. The set $\mathcal{T}_{\mathcal{S}}$ of types is generated from the set of sorts (or *basic types*) by the constructor $\rightarrow$ for *functional types*:

$$\mathcal{T}_{\mathcal{S}} := \mathcal{S} \mid (\mathcal{T}_{\mathcal{S}} \rightarrow \mathcal{T}_{\mathcal{S}})$$

In the following, we use σ, τ and ρ to denote types. *Type declarations* are expressions of the form $\sigma_1 \times \ldots \times \sigma_n \to \sigma$, where $\sigma_1, \ldots, \sigma_n, \sigma$ are types. Types occurring before the arrow are called input types, while the type occurring after the arrow is called the output type. The latter is assumed to be a sort. Type declarations are not types, although they are used for typing purposes.

2.2 Signature

We are given a set of function symbols which are meant to be algebraic operators, equipped with a fixed number n of arguments (called the *arity*) of respective types $\sigma_1, \ldots, \sigma_n$, and an output type (thereafter assumed to be a sort) σ:

$$\mathcal{F} = \bigcup_{\sigma_1, \ldots, \sigma_n, \sigma} \mathcal{F}_{\sigma_1 \times \ldots \times \sigma_n \to \sigma}$$

We will assume that there are finitely many symbols of a given output type σ.

2.3 Terms

The set of *untyped terms* is generated from a denumerable set $\mathcal{X}$ of variables according to the grammar:

$$\mathcal{T} := \mathcal{X} \mid (\lambda \mathcal{X}.\mathcal{T}) \mid \mathcal{T}(\mathcal{T}) \mid \mathcal{F}(\mathcal{T}, \ldots, \mathcal{T})$$

$u(v)$ denotes the application of u to v. We write $u(v_1, \ldots, v_n)$ for $u(v_1) \ldots (v_n)$, and may sometimes identify the first order variable x with $x()$. We use $\mathcal{V}ar(t)$ for the set of free variables of t. We may assume for convenience that bound variables are all different, and are different from the free ones.

Terms are identified with finite labelled trees by considering $\lambda x.$, for each variable x, as a unary function symbol λx. *Positions* are strings of positive integers. Λ denotes the empty string (root position). The *subterm* of t at position p is denoted by $t|_p$, and we write $t \trianglerighteq t|_p$. The result of replacing $t|_p$ at position p in t by u is denoted by $t[u]_p$. We use $t[u]$ to indicate that u is a subterm of t.

Substitutions are written as in $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where t_i is assumed different from x_i. We use the letter γ for substitutions and postfix notation for their application. Remember that substitutions behave as endomorphisms defined on free variables (avoiding captures).

2.4 Typing Rules

Typing rules restrict the set of terms by constraining them to follow a precise discipline. Environments are sets of pairs of the form $x : \sigma$, where x is a variable and σ is a type. Our typing judgements are written as $\Gamma \vdash M : \sigma$ if the term M can be proved to have the type σ in the environment Γ:

$$Variables : \frac{X:\sigma \in \Gamma}{\Gamma \vdash X:\sigma} \Bigg| \frac{F:\sigma_1 \times \ldots \times \sigma_n \to \sigma \in \mathcal{F} \quad \Gamma \vdash t_1:\sigma_1 \quad \ldots \quad \Gamma \vdash t_n:\sigma_n}{\Gamma \vdash F(t_1, \ldots, t_n):\sigma} : Functions$$

$$Abstraction : \frac{\Gamma \cup \{x{:}\sigma\} \vdash M{:}\tau}{\Gamma \vdash (\lambda x{:}\sigma.M){:}\sigma\to\tau} \Bigg| \frac{\Gamma \cup \{x{:}\sigma\} \vdash M{:}\sigma\to\tau \qquad \Gamma \vdash N{:}\sigma}{\Gamma \vdash M(N){:}\tau} : Application$$

A term M has type σ in the environment Γ if $\Gamma \vdash M : \sigma$ is provable in the above inference system. A term M is typable in the environment Γ if there exists a (necessarily unique) type σ such that M has type σ in the environment Γ. A term M is typable if it is typable in some environment Γ.

2.5 η-Long β-Normal Forms

Two particular rules originate from the λ-calculus, β-reduction and η-expansion:

$$(\lambda x.v)(u) \longrightarrow_\beta u\{x \mapsto v\}$$

$$u \longrightarrow_\eta (\lambda x.u(x)) \quad if\ u : \sigma \to \tau\ and\ x : \sigma \notin \mathcal{V}ar(u)\ and\ u \text{ is not an abstraction}$$

Given a term s, we will denote by $s{\downarrow}$ its η-long β-normal form, defined as the β-normal form of its η-expansion. We will say that s is normalized when $s = s{\downarrow}$, and use $\overset{*}{\underset{\beta\eta}{\longleftrightarrow}}$ for the congruence generated by these two rules.

The use of the extensionality rule as an expansion, and the assumption that the output type of each algebraic symbol is a sort allows us to hide applications:

Lemma 1. *Normalized terms are of either one of the two forms:*
$(\lambda x.u)$, *for some normalized term* u, *or*
$F(u_1, \ldots, u_n)$, *for some* $F \in \mathcal{F} \cup \mathcal{X}$ *and normalized terms* $u_1, \ldots, u_n$.
Moreover, terms with an arrow type are headed by an abstraction.

Note that application is implicitely present in terms headed by a higher-order variable. Abstraction-free normalized terms will be called *algebraic*.

2.6 Higher-Order Rewrite Rules

A (possibly higher-order) *term rewriting system* is a set of rewrite rules $R = \{\Gamma_i : l_i \to r_i\}_i$, where l_i and r_i are normalized terms such that l_i and r_i have the same type σ_i in the environment Γ_i. Given a term rewriting system R, a normalized term s rewrites to a term t at position p with the rule $l \to r$ and the substitution γ, written $s \xrightarrow[l\to r]{p} t$, or simply $s \to_R t$, if $s|_p \longleftrightarrow^*_{\beta\eta} l\gamma$ and $t = s[r\gamma{\downarrow}]_p$. Note that t is normalized since s is. Such a term s is called *reducible*. $l\gamma$ is a *redex* in s, and t is the *reduct* of s. Irreducible terms are said to be in *normal form*. A substitution γ is in normal form if $x\gamma$ is in normal form for all x. We denote by $\to^*_R$ the reflexive, transitive closure of the rewrite relation $\to_R$, and by $\longleftrightarrow^*_R$ its reflexive, symmetric, transitive closure.

A rewrite relation $\to$ is *terminating* (or *strongly normalizing*) if all terms are strongly normalizable, in which case it is called a *reduction*. It is confluent if $s \longrightarrow^* u$ and $s \longrightarrow^* v$ implies that $u \longrightarrow^* t$ and $v \longrightarrow^* t$ for some t.

2.7 Example 1 : Sorting

The following example presents a set of rewrite rules defining the insertion algorithm for the (ascending or descending) sort of a list of natural numbers. Note that the left and right hand sides of each rule are normalized terms.

$\mathcal{S} = \{Nat, List\}$,
$\mathcal{F}_{List} = \{nil\}, \mathcal{F}_{Nat\times Nat\to Nat} = \{max, min\}, \mathcal{F}_{Nat\times List\to List} = \{cons\}$,
$\mathcal{F}_{Nat\times List\times(Nat\times Nat\to Nat)\times(Nat\times Nat\to Nat)\to List} = \{insert\}$,
$\mathcal{F}_{List\times(Nat\times Nat\to Nat)\times(Nat\times Nat\to Nat)\to List} = \{sort\}$,
$\mathcal{F}_{List\to List} = \{ascending_sort, descending_sort\}, \mathcal{X}_{Nat\times Nat\to Nat} = \{X, Y\}$.

$$\begin{array}{cc} max(0,x) \to x & min(0,x) \to 0 \\ max(x,0) \to x & min(x,0) \to 0 \\ max(s(x),s(y)) \to s(max(x,y)) & min(s(x),s(y)) \to s(min(x,y)) \end{array}$$

$$\begin{array}{l} insert(n, nil, \lambda xy.X(x,y), \lambda xy.Y(x,y)) \to cons(x, nil) \\ insert(n, cons(m,l), \lambda xy.X(x,y), \lambda xy.Y(x,y)) \to \\ \quad cons(X(n,m), insert(Y(n,m), l, \lambda xy.X(x,y), \lambda xy.Y(x,y))) \end{array}$$

$$\begin{array}{l} sort(nil, \lambda xy.X(x,y), \lambda xy.Y(x,y)) \to nil \\ sort(cons(n,l), \lambda xy.X(x,y), \lambda xy.Y(x,y)) \to \\ \quad insert(n, sort(l, \lambda xy.X(x,y), \lambda xy.Y(x,y)), \lambda xy.X(x,y), \lambda xy.Y(x,y)) \end{array}$$

$$\begin{array}{l} ascending_sort(l) \to sort(l, \lambda xy.min(x,y), \lambda xy.max(x,y)) \\ descending_sort(l) \to sort(l, \lambda xy.max(x,y), \lambda xy.min(x,y)) \end{array}$$

2.8 Rewrite Orderings

We will make intensive use of well-founded monotonic orderings for proving strong normalization properties. We will use the vocabulary of rewrite systems for orderings: *rewrite orderings* are *monotonic* orderings, and *reduction orderings* are in addition well-founded. Monotonicity of $>$ is defined as $u > v$ implies $s[u\gamma]_p > s[v\gamma]_p$ for all terms s, positions p and substitutions γ for which this makes sense. A *quasi reduction ordering* is made of a reduction ordering for its strict part, and a congruence for its equivalence. (Quasi-) reduction orderings are used to prove termination of rewrite systems by simply comparing the left and right hand sides of rules. The following results will play a key role, see [5]:

Assume $\to_1$ and $\to_2$ are well-founded orderings on sets S_1, S_2. Then $(\to_1, \to_2)_{lex}$ is a well-founded ordering on $S_1 \times S_2$.

Assume $>$ is a well-founded ordering on a set S. Then $>_{mul}$ is a well-founded ordering on the set of multisets of elements of S. It is defined as the transitive closure of the following relation on multisets (using $\cup$ for multiset union):

$$M \cup \{s\} >> M \cup \{t_1, \ldots, t_n\} \quad \text{if } s > t_i \ \forall i \in [1..n]$$

Finally, we will also make use of simplification orderings, that is rewrite orderings that have the so-called *subterm property*: any term is strictly bigger

than any of its proper subterms. Simplification orderings contain the *embedding relation* defined as the rewrite ordering generated by subterm, that is by the projection rules $f(\vec{x}) \to x_i$ for all $f \in \mathcal{F}$ and $i \in [1..arity(f)]$. Kruskal proved that embedding is a well-order of the set of terms, see [5]. This property is the key to prove that simplification orderings are well-founded.

3 Recursive Path Ordering for Algebraic λ-Terms

Our path ordering is defined in two steps. First, we define an original extension of the recursive path ordering for algebraic terms. This extension makes an essential use of the type structure. The obtained ordering is shown to be well-founded via a simple extension of Kruskal's theorem to typed structures. Then, we define an interpretation of normalized terms into the set of purely algebraic terms, before applying the previous technique. The difficulty here is in proving that the induced ordering is closed under appropriate substitutions.

3.1 Typed Recursive Path Ordering

Our typed recursive path ordering operates on algebraic terms. It uses three basic ingredients: a quasi-ordering on types, a precedence, and a status.

The *quasi-ordering on types* $\geq_{\mathcal{T}_S}$ satisfies the following properties:

1. its strict part $>_{\mathcal{T}_S}$ is well-founded
2. $\geq_{\mathcal{T}_S}$ is *compatible with the term structure*:
   for all ground normalized terms s, $s : \sigma \trianglerighteq t : \tau$ implies $\sigma \geq_{\mathcal{T}_S} \tau$.

Note that the compatibility property of the type ordering with the term structure is actually equivalent to its *compatibility with the signature* defined as:

$$\forall F : \sigma_1 \times \ldots \times \sigma_n \to \sigma \in \mathcal{F},\ \forall i \in [1..n]\ \sigma \geq_{\mathcal{T}_S} \sigma_i$$

There are many compatible orderings, including the recursive path ordering generated by a precedence on the set of sorts and the arrow operator. Compatibility is discussed in length in section 4.

The quasi-ordering on function symbols $\geq_{\mathcal{F}}$ compares symbols of the same output type only. Its strict part $>_{\mathcal{F}}$ must be well-founded. As usual, a free variable $x : \sigma$ is considered as a function symbol comparable only to itself.

The function symbols enjoy a multiset or lexicographic status. We allow for more complex statuses like right-to-left lexicographic status for our examples. See [6] for a general notion of status.

We can now define the typed recursive path ordering:
$s : \sigma = F(\vec{s}) \succ t : \tau = G(\vec{t})$ iff

1. $\sigma >_{\mathcal{T}_S} \tau$, or
2. $\sigma =_{\mathcal{T}_S} \tau$, and

(a) $F \notin \mathcal{X}$ and $s_i \succeq t$ for some s_i, or
(b) $F >_{\mathcal{F}} G$ and $s \succ t_i$ for all i, or
(c) $F = G \in Mul$ and $\vec{s} \succ_{mul} \vec{t}$, or
(d) $F = G \in Lex$ and $\vec{s} \succ_{lex} \vec{t}$ and $s \succ t_i$ for all i

The condition $F \notin \mathcal{X}$ prevents us to prove that $X(c) \succ c$. While it is not at all necessary in the context of algebraic terms, it becomes crucial in the next subsection, in which higher-order variables can be instantiated by abstractions.

Lemma 2. *$\succeq$ is transitive, and $\succ$ is irreflexive.*

Proof. The proof is by induction on the sum of the sizes of the terms involved. Comparisons involving case 1 are straightforward. Others use the same arguments as in the corresponding proofs for rpo. □

Lemma 3. *$\succ$ has the subterm property for ground terms.*

Proof. Let $s : \sigma$ be an arbitrary ground term, and $t : \tau$ be one of its strict subterms. We proceed by induction on the size of s. If $\sigma >_{\mathcal{T}_S} \tau$, we are done. Otherwise, by the compatibility property of the type ordering with the term structure, we have $\sigma =_{\mathcal{T}_S} \tau$. Since $s = F(\vec{s})$ and t is a subterm of some s_i we can conclude by induction. □

Lemma 4. *$\succ$ is stable under ground context for equal type terms.*

Proof. Similar as for rpo. □

Lemma 5. *$\succ$ is stable under ground substitutions.*

Proof. We prove that $s \succ t$ implies $s\gamma \succ t\gamma$, for all ground substitution γ by induction on $|s|+|t|$. Assume $s : \sigma = F(s_1, \ldots, s_n)$ and $t : \tau = G(t_1, \ldots, t_m)$. We are done if $\sigma >_{\mathcal{T}_S} \tau$, since types are preserved under substitution. Otherwise, we distinguish four cases:

1. $s_i \succeq t$ for some i. By induction hypothesis, $s_i\gamma \succeq t\gamma$, and hence $s\gamma \succ t\gamma$.
2. $F >_{\mathcal{F}} G$ and $s \succ t_i$ for all i. By induction, $s\gamma \succ t_i\gamma$, and therefore $s\gamma \succ t\gamma$.
3. $F = G \in Mul$ and $\vec{s} \succ_{mul} \vec{t}$. By induction, $\vec{s\gamma} \succ_{mul} \vec{t\gamma}$.
4. $F = G \in Lex$, $\vec{s} \succ_{lex} \vec{t}$ and $s \succ t_i$ for all i. By induction, $\vec{s\gamma} \succ_{lex} \vec{t\gamma}$ and $s\gamma \succ t_i\gamma$. □

We are left with the proof of well-foundedness, which follows the usual pattern. To overcome the difficulty that the signature may be infinite, we will use our hypothesis that there are finitely many symbols for a given output type σ.

Lemma 6. *Let $\{s_i\}_{i \in \mathcal{N}}$ be an infinite sequence of ground terms of the same type σ. Then, there exist two indices $i < j$ such that s_i is embedded in s_j.*

Proof. The proof is by contradiction. If there is a counterexample sequence $\{s^i : \sigma\}_i$, there must be one of minimal type. Among such sequences of minimal type σ, we choose a minimal one (in the usual sense, by comparing the size of the corresponding terms in the sequence). Since the sequence is of type σ, terms are all headed by function symbols of output type σ, and since there are finitely many of them, we can extract an infinite subsequence of terms of the form $F(\vec{u})$, all headed by some function symbol $F \in \mathcal{F}_{\sigma_1 \times \ldots \times \sigma_n \to \sigma}$ such that $\sigma \geq_{\mathcal{T}_S} \sigma_k$ for all $k \in [1..n]$ by compatibility property of the type ordering with the term structure. We now proceed to show that infinite sequences of type σ_k built up from the terms u_k satisfy the lemma. If $\sigma >_{\mathcal{T}_S} \sigma_k$, this results from our assumption that σ is minimal. Otherwise $\sigma_k =_{\mathcal{T}_S} \sigma$, and this results from the usual minimality argument. We can therefore find indices $i < j$ such that the term u_k^i is embedded in the term u_k^j for all $k \in [1..n]$. Then $F(\vec{u^i})$ is embedded in $F(\vec{u^j})$ contradicting our assumption that there was a counterexample sequence. □

Theorem 7. *$\succeq$ is a simplification ordering on ground terms whose strict part is well-founded.*

Proof. We are left to show that $\succ$ is well-founded. Assuming an infinite decreasing sequence, all terms must have the same type from some point on in the sequence, since the ordering on types is well-founded by assumption. We use then as usual the previous lemma together with the fact that simplification orderings contain the embedding relation. □

Although we have assumed that there are finitely many symbols of the same output type σ, this is not necessary. We know that Kruskal's theorem holds for any embedding relation generated by a well-order on the set of function symbols. Since rpo generated by a given precedence contains the associated embedding relation, we can then conclude by an easy modification of our argument.

3.2 Recursive Path Ordering for Normalized Terms

The ordering for normalized terms works by comparing interpretations of normalized terms. Because bound variables cannot be substituted, they will be considered as new (non-variable) symbols c_σ of their respective type σ. Abstractions are considered as operators of some input type τ (the type of the body) and output type $\sigma \to \tau$ (σ is the type of the abstracted variable). Finally, we will index all symbols by their output type, thus making the type of a term easily readable on its head. This results in the following extended signature:

$$\begin{aligned}
\mathcal{X}_\mathcal{T} &= \{x_\sigma \mid x : \sigma \in \mathcal{X}\} \\
\mathcal{F}_\mathcal{T} &= \{F_\sigma \mid F \in \mathcal{F}_{\sigma_1 \times \ldots \times \sigma_n \to \sigma}\} \\
\lambda\mathcal{F} &= \mathcal{X}_\mathcal{T} \cup \mathcal{F} \cup \bigcup_{\sigma \in \mathcal{T}_S} \{c_\sigma\} \cup \bigcup_{\sigma,\tau \in \mathcal{T}_S} \{\lambda_{\sigma \to \tau}\}
\end{aligned}$$

Note that we have a single symbol c_σ for each type σ and a single λ operator for each arrow type. Therefore if $\mathcal{F}$ has a finite set of function symbols for each output type then $\lambda\mathcal{F}$ has a finite set of function symbols for each type.

The interpretation function maps normalized terms to typed algebraic terms over the new signature:

$$\begin{aligned} \|x : \sigma\| &= x_\sigma \\ \|(\lambda x.u) : \sigma \to \tau\| &= \lambda_{\sigma\to\tau}(\|u\|\{x_\sigma \mapsto c_\sigma\}) \\ \|F(\ldots, u_i, \ldots) : \sigma\| &= F_\sigma(\ldots, \|u_i\|, \ldots) \end{aligned}$$

Before we can introduce our ordering, we need to define the precedence on the new signature. It will simply extend the precedence over the former signature, by adding new pairs for taking care of the c_σ and $\lambda_{\sigma\to\tau}$ operators. Being unary operators, abstractions may have any status. Finally, the compatibility property for the abstraction operators requires that $\sigma \to \tau \geq_{\mathcal{T}_S} \tau$. Compatibility is investigated thoroughly in section 4.

Assuming that s and t are two normalized terms such that no variable is both free and bound nor bound twice in the pair (s, t), we define:

$$s \succ_{horpo} t \text{ iff } \|s\| \succ \|t\|$$

Lemma 8. *$\succ_{horpo}$ is well-founded on ground normalized terms.*

Proof. The property is inherited from the previous ordering. □

Lemma 9. *If $s \succ_{horpo} t$ then $u[s] \succ_{horpo} u[t]$ for all contexts u and terms $s : \tau$ and $t : \tau$ such that $u[s]$ and $u[t]$ are ground normalized terms.*

Proof. Since s and t have the same type τ, $u[s]$ and $u[t]$ have the same type. We prove that $F(\ldots, s, \ldots) \succ_{horpo} F(\ldots, t, \ldots)$ for all possible F. Since the context must be ground, F cannot be a variable, resulting in two cases:

1. If F is an algebraic function symbol, then $F(\ldots, s, \ldots) \succ_{horpo} F(\ldots, t, \ldots)$ iff $\|F(\ldots, s, \ldots)\| \succ \|F(\ldots, t, \ldots)\|$, that is $F_\tau(\ldots, \|s\|, \ldots) \succ F_\tau(\ldots, \|t\|, \ldots)$, where τ is the output type of F. Since $\|s\| \succ \|t\|$ holds by assumption that $s \succ_{horpo} t$ and definition of $\succ_{horpo}$, the result holds by monotonicity of $\succ$ with respect to ground contexts.
2. Otherwise, let F be λx, for some x of type σ. Since $s \succ_{horpo} t$ iff $\|s\| \succ \|t\|$, then $\|s\|\{x_\sigma \mapsto c_\sigma\} \succ \|t\|\{x_\sigma \mapsto c_\sigma\}$ by lemma 5.
   By definition, $\lambda x.s \succ_{horpo} \lambda x.t$ iff $\|\lambda x.s\| \succ \|\lambda x.t\|$ iff $\lambda_{\sigma\to\tau}(\|s\|\{x \mapsto c_\sigma\}) \succ \lambda_{\sigma\to\tau}(\|t\|\{x \mapsto c_\sigma\})$. We therefore conclude by stability of $\succ$ under ground contexts.

□

Lemma 10. *$\succ_{horpo}$ is stable under ground normalized substitutions.*

Proof. Assuming that $s : \sigma$ and $t : \tau$ are normalized, we need to show that $s \succ_{horpo} t$ implies $s\gamma\downarrow \succ_{horpo} t\gamma\downarrow$ for all ground normalized substitutions γ. This amounts to show that $s' = \|s\| \succ \|t\| = t'$ implies $\|s\gamma\downarrow\| \succ \|t\gamma\downarrow\|$. This cannot be proved directly due to the replacement of bound variables by new function symbols (the c_σ) in the interpretation. We will therefore prove a more general statement, namely $\|s\|\psi_1 \succ \|t\|\psi_2$ implies $\|s\gamma\downarrow\|\psi_1 \succ \|t\gamma\downarrow\|\psi_2$, where ψ_1 and

ψ_2 are (well-typed) substitutions taking their value in $\bigcup_{\sigma \in \mathcal{T}_S} \{c_\sigma\}$, and such that if $X_{\sigma'} \in \mathcal{D}om(\psi_1) \cup \mathcal{D}om(\psi_2)$ (for some type σ') then $X \notin \mathcal{D}om(\gamma)$. We proceed by induction on $|s'| + |t'|$.

The statement holds if $\sigma >_{\mathcal{T}_S} \tau$, since types are preserved under substitution, normalization and interpretation.

Otherwise, let $s' = F_\sigma(s'_1, \ldots, s'_n)$ and $t' = G_\sigma(t'_1, \ldots, t'_m)$. We distinguish five cases:

1. $s'\psi_1 \succ t'\psi_2$ by case 2a, that is $s'_i\psi_1 \succeq t'\psi_2$ for some subterm s'_i of s'. Since the head symbol of $s'\psi_1$ cannot be a variable by definition of the ordering, there are two possibilities: (i) $s = F(s_1, \ldots, s_n)$, where $F \in \mathcal{F}$ or $F \in \mathcal{X}$ and $F_\sigma\psi_1 = c_\sigma$; or else (ii) s is an abstraction $\lambda x.s_1$, with $x : \sigma'$.
   i. In the first case, we have $s'\psi_1 = F_\sigma\psi_1(s'_1\psi_1, \ldots, s'_n\psi_1)$, where $s'_i = \|s_i\|$. By induction hypothesis, $s'_i\psi_1 \succ t'$ implies $\|s_i\gamma\downarrow\ \|\psi_1 \succ \|t\gamma\downarrow\ \|\psi_2$. Since $\|F(s_1, \ldots, s_n)\gamma\downarrow\ \|\psi_1 = F_\sigma\psi_1(\|s_1\gamma\downarrow\ \|\psi_1, \ldots, \|s_n\gamma\downarrow\ \|\psi_1)$, the result holds.
   ii. In the second case, $s'\psi_1 = \lambda_\sigma(s'_1\psi_1)$, where $s'_1 = \|s_1\|\{x_{\sigma'} \mapsto c_{\sigma'}\}$. Then $s'_1\psi_1$ can be written as $\|s_1\|\psi'_1$, with $\psi'_1 = \{x_{\sigma'} \mapsto c_{\sigma'}\} \cup \psi_1$. Note that x cannot appear in the domain of γ since it is bound in s. Likewise, $x_{\sigma'}$ cannot appear in the domain of ψ_1. Therefore, by induction hypothesis, $s'_1\psi_1 = \|s_1\|\psi'_1 \succ t'\psi_2$ implies $\|s_1\gamma\downarrow\ \|\psi'_1 \succ \|t\gamma\downarrow\ \|\psi_2$. The expected result follows, since $\|(\lambda x.s_1)\gamma\downarrow\ \|\psi_1 = \lambda_\sigma((\|s_1\gamma\downarrow\ \|\{x_{\sigma'} \mapsto c_{\sigma'}\})\psi_1) = \lambda_\sigma(\|s_1\gamma\downarrow\ \|\psi'_1)$.
2. $F_\sigma\psi_1 >_{\lambda\mathcal{F}} G_\sigma\psi_2$ and $s'\psi_1 \succ t'_i\psi_2$ for all i. There are again two cases.
   i. If $G_\sigma \neq \lambda_\sigma$, then, by induction, $\|s\gamma\downarrow\ \|\psi_1 \succ \|t_i\gamma\downarrow\ \|\psi_2$. Since $\|G(t_1, \ldots, t_m)\gamma\downarrow\ \|\psi_2 = G_\sigma\psi_2(\|t_1\gamma\downarrow\ \|\psi_2, \ldots, \|t_m\gamma\downarrow\ \|\psi_2)$, the result follows.
   ii. If $G_\sigma = \lambda_\sigma$ then, $t'\psi_2 = \lambda_\sigma(t'_1\psi_2)$, where $t'_1 = \|t_1\|\{x_{\sigma'} \mapsto c_{\sigma'}\}$ for some $x : \sigma'$. Since $t'_1\psi_2$ can be written as $\|t_1\|\psi'_2$ (with $\psi'_2 = \{x_{\sigma'} \mapsto c_{\sigma'}\} \cup \psi_2$), by induction hypothesis, $\|s\gamma\downarrow\ \|\psi_1 \succ \|t_1\gamma\downarrow\ \|\psi'_2$, and since $\|(\lambda x.t_1)\gamma\downarrow\ \|\psi_2 = \lambda_\sigma(\|t_1\gamma\downarrow\ \|\psi'_2)$, the result holds.
3. $F_\sigma = G_\sigma = \lambda_\sigma$ and $s'_1\psi_1 \succ t'_1\psi_2$. Then $s'\psi_1 = \lambda_\sigma(s'_1\psi_1)$, where $s'_1 = \|s_1\|\{x_{\sigma'} \mapsto c_{\sigma'}\}$ for some $x : \sigma'$, and $t'\psi_2 = \lambda_\sigma(t'_1\psi_2)$, where $t'_1 = \|t_1\|\{y_{\sigma'} \mapsto c_{\sigma'}\}$ for some $y : \sigma'$. Since $s'_1\psi_1$ can be written as $\|s_1\|\psi'_1$ (with $\psi'_1 = \{x_{\sigma'} \mapsto c_{\sigma'}\} \cup \psi_1$) and $t'_1\psi_2$ can be written as $\|t_1\|\psi'_2$ (with $\psi'_2 = \{y_{\sigma'} \mapsto c_{\sigma'}\} \cup \psi_2$), by induction hypothesis, $\|s_1\gamma\downarrow\ \|\psi'_1 \succ \|t_1\gamma\downarrow\ \|\psi'_2$. Since $\|(\lambda x.s_1)\gamma\downarrow\ \|\psi_1 = \lambda_\sigma(\|s_1\gamma\downarrow\ \|\psi'_1)$ and $\|(\lambda y.t_1)\gamma\downarrow\ \|\psi_2 = \lambda_\sigma(\|t_1\gamma\downarrow\ \|\psi'_2)$, the result holds.
4. $F_\sigma\psi_1 = G_\tau\psi_2 \in Mul$ and $s'\vec{\psi}_1 \succ_{mul} t'\vec{\psi}_2$. By induction, the result holds.
5. $F_\sigma\psi_1 = G_\tau\psi_2 \in Lex$ and $s'\vec{\psi}_1 \succ_{lex} t'\vec{\psi}_2$ and $s'\psi_1 \succ t'_i\psi_2$ for all i. By induction, the result follows similarly.

Note that the cases in which the head symbol may change, namely when the head symbol is a higher-order variable, are always solved by type, therefore avoiding potential problems. □

Theorem 11. *If $l \succ r$ for all rules $l \to r$ in R then R is terminating.*

Proof. First, terms in a derivation can always be considered as ground, therefore allowing us to use the previous properties. Assume now that $s \xrightarrow[l \to r]{p} t$, where s and t are ground normalized terms. Then, $s|_p =_{\beta\eta} l\gamma$, and $t = s[r\gamma\downarrow]_p$. Since $\longrightarrow_{\beta\eta}$ is confluent, and s is normalized, then $s|_p = l\gamma\downarrow$. By assumption, $l \succ_{horpo} r$, hence $l\gamma\downarrow \succ_{horpo} r\gamma\downarrow$ by lemma 10, and $s = s[l\gamma\downarrow]_p \succ_{horpo} s[r\gamma\downarrow]_p = t$ by lemma 9. Since $\succ_{horpo}$ is well-founded, R is therefore terminating. □

The above theorem is aimed at proving termination of higher-order rewrite rules operating on normalized terms via higher-order pattern matching. The use of first-order pattern matching does not change the situation very much, as long as rewriting operates on normalized terms.

4 Compatible Type Orderings

The goal of this section is to provide a sufficient condition ensuring the compatibility property of the type ordering with the term structure, that is $s : \sigma \trianglerighteq t : \tau$ implies $\sigma \geq_{\mathcal{TS}} \tau$ for all ground normalized terms s and t. We will then exhibit two actual orderings which satisfy these conditions and hence can be used as a component of our ordering on higher-order terms.

Definition 12. The quasi-ordering on types $\geq_{\mathcal{TS}}$ is *compatible with the type structure* if

(i) $\forall f : \sigma_1 \times \ldots \times \sigma_n \to \sigma \in \mathcal{F}$, $\forall i \in [1..n]$, $\sigma \geq_{\mathcal{TS}} \sigma_i$,

(ii) $\sigma \to \tau \geq_{\mathcal{TS}} \sigma$ and $\sigma \to \tau \geq_{\mathcal{TS}} \tau$.

Note that conditions (i) and (ii) are indeed necessary. For the first part of (ii), consider the example $\lambda x.x(a)$, in which a is a constant of type $\tau >_{\mathcal{TS}} \sigma$, and $x : \tau \to \sigma$.

Lemma 13. *Let $\geq_{\mathcal{TS}}$ be a quasi-ordering on types compatible with the type structure. Then, for every ground normalized term $s : \sigma$ and for every ground normalized subterm $t : \tau$, we have $\sigma \geq_{\mathcal{TS}} \tau$.*

Proof. We prove the more general property that $\sigma \geq_{\mathcal{TS}} \tau$ for every normalized term $s : \sigma$ and every subterm $t : \tau$ not below a free variable (free variables may not satisfy the compatibility property). We proceed by induction on the size of the term. The property trivially holds if t is a constant or is headed by a free variable. Otherwise, there are two possible cases:

1. $s = F(s_1, \ldots, s_n)$, with $s_i : \sigma_i$. By compatibility of the type ordering with the type structure, $\sigma \geq_{\mathcal{TS}} \sigma_i$. By induction hypothesis, for every subterm $t : \tau$ (not headed by a free variable) of some $s_i : \sigma_i$, we have $\sigma_i \geq_{\mathcal{TS}} \tau$, and hence $\sigma \geq_{\mathcal{TS}} \tau$. Note that we have considered all possible subterms (not headed by a free variable) of s.

2. $s = \lambda x.u$, with $\sigma = \alpha \to \beta$. By compatibility $\sigma \geq_{\mathcal{T}_S} \alpha$ and $\sigma \geq_{\mathcal{T}_S} \beta$. By induction hypothesis for every subterm $t : \tau$ of $u : \beta$ not below a free variable, $\beta \geq_{\mathcal{T}_S} \tau$ and hence $\sigma \geq_{\mathcal{T}_S} \tau$. The only new free variable of u (wrt. s) is x. Therefore, we need to ensure that for all subterms $t : \tau$ below some occurrence of x, we also have $\sigma \geq_{\mathcal{T}_S} \tau$. Let α be $\alpha_1 \to \ldots \alpha_n \to \alpha_{n+1}$, with $n \geq 0$. Assume there is a subterm of u of the form $x(u_1, \ldots, u_n)$. By compatibility, $\sigma \geq_{\mathcal{T}_S} \alpha \geq_{\mathcal{T}_S} \alpha_i$ for all i and by induction hypothesis for every subterm $t : \tau$ (not below a free variable) of some u_i $\alpha_i \geq_{\mathcal{T}_S} \tau$ and hence $\sigma \geq_{\mathcal{T}_S} \tau$. □

There is a tradeoff between the granularity and the compatibility of the ordering on types. This is why we provide with two different orderings satisfying the compatibility properties. The first is a recursive path ordering on type expressions. This is a fine grain ordering, thus compatibility may therefore be hard to achieve. The second ordering identifies many types, allowing us to ensure compatibility easily, but its strict part compares fewer types.

4.1 Recursive Path Ordering on Types

As already noticed in the context of algebraic terms, the recursive path ordering itself is a good candidate for the type ordering. Condition (ii) is always satisfied (the comparison is strict), and condition (i) is easily checked when the signature $\mathcal{F}$ is finite. The signature to be considered for the type ordering is the set of basic types plus the arrow operator. Usually, it is better to take $\to$ smaller than any basic type, although there is no obligation. The only requirement is to provide a precedence $\geq_{\mathcal{S}}$ on $\mathcal{S} \cup \{\to\}$, and a status for $\to$. The obtained ordering is quite powerful, hence it will allow us to compare many terms by type.

Example 1 : Sorting
We use the following precedences:
$List >_{\mathcal{S}} Nat$,
$\{max, min\} >_{\mathcal{F}} \{0, s\}$,
$insert >_{\mathcal{F}} cons$, $sort >_{\mathcal{F}} \{nil, insert\}$,
$\{ascending_sort, descending_sort\} >_{\mathcal{F}} sort$,
$insert \in Lex$ (right to left).

Example 2 : Mapcar
$\mathcal{S} = \{Nat, List\}$,
$\mathcal{F}_{List} = \{nil\}, \mathcal{F}_{Nat \times List \to List} = \{cons\}, \mathcal{F}_{(Nat \to Nat) \times List \to List} = \{map\}$,
$\mathcal{X}_{Nat \to Nat} = \{X\}$

$$map(\lambda x.X(x), nil) \to nil$$
$$map(\lambda x.X(x), cons(n, l)) \to cons(X(n), map(\lambda x.X(x), l))$$

The termination proof may use the following precedences:
$List >_{\mathcal{S}} Nat$, $List >_{\mathcal{S}} \to$, $map >_{\mathcal{F}} cons$, $map \in Mul$.

Example 3 : Maplist
$\mathcal{S} = \{Nat, List\}$,

$\mathcal{F}_{List} = \{nil\}, \mathcal{F}_{Nat \times List \to List} = \{cons\}, \mathcal{F}_{(Nat \to Nat) \times List \to List} = \{fcons\},$
$\mathcal{F}_{List \times Nat \to List} = \{fmap\},$
$\mathcal{X}_{Nat \to Nat} = \{X\}$

$$fmap(nil, n) \ \to \ nil$$
$$fmap(fcons(\lambda x.X(x), L), n) \ \to \ cons(X(n), fmap(L, n))$$

The termination proof may use the following precedences:

$$List \ >_{\mathcal{S}} \ Nat, \quad List \ >_{\mathcal{S}} \to, \qquad fmap \ >_{\mathcal{F}} \ cons, \qquad fmap \in Mul.$$

4.2 Sort Ordering

The second ordering is again generated by a precedence $\geq_{\mathcal{S}}$ on the set of basic types. It compares type expressions according to the maximal basic type occurring in them, that is

$$\sigma \geq_{\mathcal{T}_{\mathcal{S}}} \tau \quad \text{iff} \quad max(\sigma) \geq_{\mathcal{S}} max(\tau)$$

where $max(\sigma)$ is the maximal (wrt. $\geq_{\mathcal{S}}$) basic type occuring in σ.

Example 4 : Maplist_to_list

$\mathcal{S} = \{Nat, List\},$
$\mathcal{F}_{List} = \{nil\}, \mathcal{F}_{Nat \times List \to List} = \{cons\}, \mathcal{F}_{(List \to Nat) \times List \to List} = \{fcons\},$
$\mathcal{F}_{List \times Nat \to List} = \{fmap\}.$

$$fmap(nil, s) \ \to \ nil$$
$$fmap(fcons(\lambda x.X(x), L), s) \ \to \ cons(X(s), fmap(L, s))$$

The termination proof may use the following precedences:

$$List \ >_{\mathcal{S}} \ \{Nat, \to\}, \qquad fmap \ >_{\mathcal{F}} \ cons, \qquad fmap \in Mul$$

Note that since $List \to Nat$ is necessarily strictly bigger than $List$ for a recursive path ordering, $fcons$ would not fulfil the compatibility property with respect to any recursive path ordering on types.

5 Discussion

Three recent papers and a phd thesis discuss termination of higher-order rewrite rules, among which two give methods based on orderings for higher-order terms. In [10], the user is responsible for proving the monotonicity property of the ordering used, in the form of proof obligations that he or she has to verify. Besides, the resulting ordering is still weak, assuming in particular that left-hand sides are patterns in the sense of Miller. The method described in [3] is much more powerful, since it allows to prove termination of cut-elimination calculi, to the price of loosing control on the amount of verifications to be performed. On the other hand, [9] describes a computable ordering in the same spirit as ours, based on a first-order interpretation of terms in η-long β-normal form. According to [10] who give a counter example, this ordering may not be well-founded, but

the same authors believe that the ordering defined in [8] is correct. It is also restricted to patterns in left-hand sides, a restriction that we do not need, but is equipped with a powerful notion of higher-order subterm, allowing to state that $v(x,u)$ is a subterm of $rec(S(x),u,\lambda yz.v)$. However, this notion of subterm is not enough to deal with the following example of primitive recursion of higher type over the natural numbers in Peano notation:

$$\begin{aligned} rec(0,u,\lambda yz.v(y,z)) &\to u \\ rec(S(x),u,\lambda yz.v(y,z)) &\to v(x,rec(x,u,\lambda yz.v(y,z))) \end{aligned}$$

Indeed, our ordering cannot deal with this example either, because our notion of compatibility is defined in terms of the first-order notion of subterm: given a type γ, the higher-order variable v has type $Nat \times \gamma \to \gamma$, and Gödel's recursor rec has type $Nat \times \gamma \times (Nat \times \gamma \to \gamma) \to \gamma$. Since γ cannot be bigger than $Nat \times \gamma \to \gamma$ in the ordering on types, compatibility requires all types to be equal, and the variable v cannot be made smaller by type comparison. The very reason of this failure is that we implicitely consider $\lambda yz.v(x,y)$ as a subterm of $rec(x,u,\lambda yz.v(y,z))$. Applying $\lambda yz.v(x,y)$ to inputs of the appropriate type as in the definition of the higher-order subterm, results in a term of type γ, and compatibility becomes satisfied. The use of higher-order subterms therefore results in a sharper notion of compatibility of an operator f in terms of the output type (instead of the type itself) of each of its inputs.

On the other hand, our ordering is already powerful enough to treat a variety of examples, as we have shown in section 4. For all these examples, the use of type information was essential. As a matter of fact, even the first example, *sorting*, cannot be proved terminating by using the orderings of [9, 8], which fail to check the second *insert* rule.

6 Conclusion

We have described an ordering for higher-order terms in η-long β-normal form, which enables us to automatically prove termination properties of higher-order rewrite rules in this setting. This ordering extends two distinct orderings to typed terms, an ordering on function symbols and an ordering on types. As a result of this structure similar to that of the recursive path ordering for first-order terms, the ordering is easy to implement and to use in practice, and indeed, we have demonstrated its applicability by giving several practical examples. Improving over previous attempts, the ordering does not assume that left-hand sides of rules are patterns. It requires instead that the ordering on types is compatible with the signature, a quite restrictive property indeed.

However, we believe that we can do a lot better by incorporating to our ordering a notion of higher-order subterm as the one proposed in [8]. For this, we need to show that the embedding relation defined by an appropriate subterm ordering for higher-order terms is a well-order: this would allow us to generalize the present work quite smoothly. We are currently progressing in this direction which should yield what we think should be the true generalization of the recursive path ordering to this higher-order setting.

References

1. Henk Barendregt. *Handbook of Theoretical Computer Science*, volume B, chapter Functional Programming and Lambda Calculus, pages 321–364. North-Holland, 1990. J. van Leeuwen ed.
2. Henk Barendregt. *Handbook of Logic in Computer Science*, chapter Typed lambda calculi. Oxford Univ. Press, 1993. eds. Abramsky et al.
3. Jaco Van de Pol and Helmut Schwichtenberg. Strict functional for termination proofs. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications, Edinburgh, Great Britain*, 1995.
4. Nachum Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, March 1982.
5. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–309. North-Holland, 1990.
6. Maribel Fernández and Jean-Pierre Jouannaud. Modular termination of term rewriting systems revisited. In Egidio Astesiano, Gianni Reggio, and Andrzej Tarlecki, editors, *Recent Trends in Data Type Specification*, volume 906 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995. Refereed selection of papers presented at ADT'94.
7. Deepak Kapur and Han Tao Zhang. An overview of the rewrite rule laboratory (RRL). In *Proc. 3rd Rewriting Techniques and Applications, Chapel Hill, LNCS 355*, pages 559–563. Springer-Verlag, 1989.
8. Carlos Loría-Sáenz. *A Theoretical Framework for Reasoning about Program Construction based on Extensions of Rewrite Systems*. PhD thesis, Fachbereich Informatik der Universität Kaiserslautern, 1993.
9. Carlos Loría-Sáenz and Joachim Steinbach. Termination of combined (rewrite and λ-calculus) systems. In *Proc. 3rd Int. Workshop on Conditional Term Rewriting Systems, Pont-à-Mousson, LNCS 656*, volume 656 of *Lecture Notes in Computer Science*, pages 143–147. Springer-Verlag, 1992.
10. Olav Lysne and Javier Piris. A termination ordering for higher order rewrite systems. In *Proc. 6th Rewriting Techniques and Applications, Kaiserslautern, LNCS 914*, Kaiserslautern, Germany, 1995.
11. Claude Marché. Normalised rewriting and normalised completion. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, Paris, France, July 1994. IEEE Comp. Soc. Press.
12. Tobias Nipkow. Higher order critical pairs. In *Proc. IEEE Symp. on Logic in Comp. Science*, Amsterdam, 1991.
13. Pilar Nivela and Robert Nieuwenhuis. Practical results on the saturation of full first-order clauses: Experiments with the saturate system. (system description). In C. Kirchner, editor, *5th International Conference on Rewriting Techniques and Applications*, LNCS 690, Montreal, Canada, June 16–18, 1993. Springer-Verlag.

This article was processed using the LaTeX macro package with LLNCS style