Rewrite Orderings for Higher-Order Terms in η-Long β-Normal Form and the Recursive Path Ordering*

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.

Key words: *higher-order rewriting*; *typed lambda calculus*; *termination orderings.*

1 Introduction

Higher-order rewrite rules are used in various programming languages and logical systems with two different meanings. Some functional languages like ML or type theories like the calculus of inductive constructions use higher-order rewrite rules to define functions or recursors by first-order pattern matching. In this setting termination is known to be satisfied when all rules follow a generalized form of a primitive recursive schema of higher type [9, 1, 10]. 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 [16]. On the negative side, this framework lacks adequate termination proof methods.

Computer systems like RRL [11], *Saturate* [17] or CiME [15] 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 [5]. Its principle is to generate recursively an ordering on terms from a user-defined ordering on function symbols called precedence.

* This work was partly supported by the ESPRIT Basic Research Action CCL and the EU Human Capital and Mobility Network *Console*. A preliminary version is published in the proceedings of the 7th International Conference on Rewriting Techniques and Applications, Lecture Notes In Computer Science 1103, Springer Verlag.
** This work results from a stay of the author at the Technical University of Catalonia.

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 [13, 14, 4]. 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 η-long β-normal form.

Our contribution is precisely the definition of a recursive path ordering for higher-order terms operating on terms in η-long β-normal form. This ordering extends, on one hand a precedence on the function symbols, on the other hand a well-founded ordering on the type structure. Terms are compared by type first, then by head function symbol, before the comparison can proceed recursively on the arguments. 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 or by its basic output type. Several examples are carried out. Note that, as a particlular case, our ordering can be used to prove termination of many-sorted first-order term rewiriting systems (see [18] for an earlier work on this problem).

The ordering works on *type compatible terms*, i.e. terms for which subterms are compatible and have a type smaller than or equal to (wrt. the ordering on types) the type of the term. Therefore, the ordering allows us to prove termination of the higher-order rewrite systems over compatible terms. For this reason, we provide sufficient conditions ensuring the compatibility property of the terms, which forces us to impose some restrictions on the ordering on types wrt. the type structure of the signature. We also show that even for some non-compatible terms the termination is assured by the termination of the rewrite system on compatible terms.

The idea of extending a recursive path ordering to higher-order terms in η-long β-normal form was first explored in [13]. The authors restricted their study to patterns in the sense of Miller, and this restriction survived in subsequent work [12, 14]. 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 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 [12]. We believe that its use in our definition would overcome the limitations of our current proposal.

The framework we use is described in Section 2. In Section 3 we extend the usual definitions on rewrite orderings to higher-order terms. The ordering is defined and studied in Section 4. Section 5 is devoted to analyze the compatibility property and Section 6 presents some actual orderings on types and several examples of application. A comparison with previous work is given in Section 7 and conclusions and future work in Section 8.

We expect the reader to be familiar with the basic concepts and notations of term rewriting systems [7] and typed lambda calculi [2, 3].

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 $\sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma$, 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. We will denote by $O(\sigma)$ the output type (sort) of σ. Type declarations are not types, although they are used for typing purposes. The notion of input and output types will also be used for types, considering that the output type of a sort is the type itself and for a type $\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma$, where σ is a sort, again σ is the output type and all others are the input types.

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:

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

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, and $\mathcal{BV}ar(t)$ for its set of bound variables. We may assume for convenience (and without further notice) that bound variables in a term are all different, and are different from the free ones.

Terms are identified with finite labeled trees by considering $\lambda x.$, for each variable x, as a unary function symbol λx. A *context* is a term with a hole. If t is a context and u a term then $t[u]$ is the term resulting of placing u in the hole of t. A term (context) is *ground* if it contains no free variables . *Positions* are strings of positive integers. Λ and $\cdot$ denote respectively the empty string (root position) and the concatenation of strings. The latter may sometimes be omitted. 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. We use $\overline{s}$ to denote a list of terms $s_1, \ldots, s_n$, and it will be used ambiguously as a sequence or as a multiset of terms depending on the context.

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:	$\dfrac{X : \sigma \in \Gamma}{\Gamma \vdash X : \sigma}$
Functions:	$\dfrac{F : \sigma_1 \times \ldots \times \sigma_n \to \sigma \in \mathcal{F} \quad \Gamma \vdash t_1 : \sigma_1 \ \ldots \ \Gamma \vdash t_n : \sigma_n}{\Gamma \vdash F(t_1, \ldots, t_n) : \sigma}$
Abstraction:	$\dfrac{\Gamma \cup \{x : \sigma\} \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.M) : \sigma \to \tau}$
Application:	$\dfrac{\Gamma \cup \{x : \sigma\} \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M(N) : \tau}$

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 Higher-Order Rewrite Rules

Three particular equations originate from the λ-calculus, α-, β- and η-equality:

$$\boxed{\begin{array}{lll} \lambda x.v & =_\alpha \lambda y.v\{x \mapsto y\} & \text{if } y \notin \mathcal{BV}ar(v) \cup \mathcal{V}ar(v) \\ (\lambda x.v)(u) & =_\beta \ u\{x \mapsto v\} & \\ \lambda x.u(x) & =_\eta \quad u & \end{array}}$$

As usual, we do not distinguish terms which are α-equivalent. We use $\overset{*}{\underset{\beta\eta}{\longleftrightarrow}}$ for the congruence generated by the latter two equations. The first is easily oriented to give the β-reduction rule below. For the second, there is a choice, either from left to right yielding η-reduction, or from right to left yielding η-expansion. The second choice has well-known advantages [8], but η-expansion must be restricted so as to ensure termination, and, as a result, it is not a congruence. This is why we need to spell out in which context an η-expansion applies:

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

$$C[u]_p \longrightarrow_\eta C[\lambda x.u(x)]_p \ \textbf{if} \begin{cases} u : \sigma \to \tau \\ x \notin \mathcal{V}ar(u) \\ u \text{ is not an abstraction} \\ C[u]|_q \text{ is not an application in case } p = q \cdot 1 \end{cases}$$

The simply typed λ-calculus is confluent and terminating with respect to β-reductions, and with respect to β-reductions and η-expansions as well. We write $s\!\downarrow_\beta$ for the unique β-normal form of a term, and $u \updownarrow^\eta_\beta$ or simply $s\!\downarrow$ for its unique (up to α-equivalence) normal form with respect to β-reductions and η-expansions, also called η-long β-normal form. We will say that s is *normalized* when $s = s\!\downarrow$. Since the two rules are confluent, η-long β-normal forms can be obtained by β-normalization followed by η-expansion. As a consequence, these normal forms satisfy the following property, which justifies the terminology:

Lemma 1. *η-long β-normal forms are of the form $\lambda x_1 \ldots x_m.F(u_1, \ldots, u_n)$, for some $m \geq 0$, $F \in \mathcal{F} \cup \mathcal{X}$ and terms $u_1, \ldots, u_n$ in η-long β-normal form themselves, such that $F(u_1, \ldots, u_n)$ is of basic type.*

Corollary 2. *Let s, t be two normalized terms such that $s|_p$ and t have the same type. Then $s[t]_p$ is in η-long β-normal form.*

The use of the extensionality rule as an expansion, and the assumption that the output type of each algebraic symbol is a basic type allows us to hide applications except when $F \in \mathcal{X}$. Abstraction-free normalized terms will be called *algebraic*. Algebraic terms are first-order when they do not contain higher-order variables. One advantage of η-expansions is that higher-order algebraic terms look and sometimes behave as first-order ones.

A (possibly higher-order) *term rewriting system* is a set of rewrite rules $R = \{\Gamma_i \vdash l_i \rightarrow r_i : \sigma_i\}_i$, 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 $l \rightarrow r$ and the substitution γ, written $s \xrightarrow[l\rightarrow r]{p} t$, or simply $s \rightarrow_R t$, if $s|_p \longleftrightarrow^*_{\beta\eta} l\gamma$ and $t = s[r\gamma\!\downarrow]_p$. By Corollary 2 t is normalized since s is. Such a term s is called *reducible* (with respect to R). $l\gamma$ is a *redex* in s, and t is the *reduct* of s. Irreducible terms are said to be in R-*normal form*. A substitution γ is in R-normal form if $x\gamma$ is in R-normal form for all x. Note that if the applied rule $l \rightarrow r$ has non-basic type σ then we have to require $s|_p$ to have type σ too, since otherwise $s|_p$ and $r\gamma\!\downarrow$ may have different types. If the type of the rule is basic then this restriction is superfluous as it always holds.

We denote by $\rightarrow^*_R$ the reflexive, transitive closure of the rewrite relation $\rightarrow_R$, and by $\longleftrightarrow^*_R$ its reflexive, symmetric, transitive closure.

A rewrite relation $\rightarrow$ 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.

2.6 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.

$\mathcal{S} = \{Nat, List\}$,
$\mathcal{F}_{List} = \{nil\}, \mathcal{F}_{Nat\times Nat\rightarrow Nat} = \{max, min\}, \mathcal{F}_{Nat\times List\rightarrow List} = \{cons\}$,
$\mathcal{F}_{Nat\times List\times(Nat\times Nat\rightarrow Nat)\times(Nat\times Nat\rightarrow Nat)\rightarrow List} = \{insert\}$,
$\mathcal{F}_{List\times(Nat\times Nat\rightarrow Nat)\times(Nat\times Nat\rightarrow Nat)\rightarrow List} = \{sort\}$,
$\mathcal{F}_{List\rightarrow List} = \{ascending\_sort, descending\_sort\}, \mathcal{X}_{Nat\times Nat\rightarrow Nat} = \{X, Y\}$.

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

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

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

$$
\begin{array}{l}
ascending\_sort(l) \rightarrow sort(l, \lambda xy.min(x,y), \lambda xy.max(x,y)) \\
descending\_sort(l) \rightarrow sort(l, \lambda xy.max(x,y), \lambda xy.min(x,y))
\end{array}
$$

2.7 Rewrite Orderings

We will make intensive use of well-founded monotonic (possibly total) orderings for proving strong normalization properties. We will use the vocabulary of rewrite systems for orderings. An ordering, usually denoted by $>$, is an irreflexive and transitive relation, and a quasi-ordering, usually denoted by $\geq$, is a reflexive transitive relation. A quasi-ordering $\geq$ can be split in an ordering $>$ for its strict part, defined by $s > t$ iff $s \geq t \wedge \neg(t \geq s)$, and an equivalence for the rest, i.e. when $s \geq t$ and $t \geq s$. *Rewrite orderings* are *monotonic* orderings, and *reduction orderings* are in addition well-founded. A quasi-ordering is said to be well-founded when its strict part is 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. Totality (on ground terms) means that any two (ground) terms are comparable. (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 [7]:

Assume $\geq_1$ and $\geq_2$ are well-founded quasi-orderings on sets S_1, S_2. Then $(\geq_1, \geq_2)_{lex}$ is a well-founded quasi-ordering on $S_1 \times S_2$.

Assume $\geq_1$ and $\geq_2$ are well-founded quasi-orderings on sets S_1, S_2. Then $(s_1, s_2)(\geq_1, \geq_2)_{mon}(t_1, t_2)$ iff $s_1 \geq_1 t_1$ and $s_2 \geq_2 t_2$ is a well-founded quasi-ordering on $S_1 \times S_2$.

Assume $\geq$ is a well-founded quasi-ordering on a set S. Then $\geq_{mul}$ is a well-founded quasi-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):

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

Note that the equivalence generated by $\geq_{mul}$ is obtained by taking the reflexive transitive closure of case two of the above definition when applied to equivalent terms s and t.

Finally, we will also make use of simplification orderings, that are rewrite orderings possessing 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 $f(\overline{x}) \to x_i$ 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 [7]. This property is the key to prove that simplification orderings are well-founded [5]. The most popular simplification ordering is the *recursive path ordering* of Dershowitz [5]:

Assume $>_{\mathcal{F}}$ is a well-founded ordering on $\mathcal{F} = Mul \uplus Lex$. Then the following ordering on terms, called recursive path ordering, is a reduction ordering:

$s = f(\overline{s}) \succeq_{rpo} t = g(\overline{t})$ iff

1. $s_i \succeq_{rpo} t$ for some i
2. $f >_{\mathcal{F}} g$ and $s \succ_{rpo} t_i$ for all i

3. $f = g \in Lex$ and $\overline{s}(\succeq_{rpo})_{lex}\overline{t}$ and $s \succ_{rpo} t_i$ for all i
4. $f = g \in Mul$ and $\overline{s}(\succeq_{rpo})_{mul}\overline{t}$

where $s \succ_{rpo} t$ iff $s \succeq_{rpo} t$ and $t \not\succeq_{rpo} s$.

Here Mul and Lex defines the *status* of the function symbols, which means the way the arguments should be compared recursively. In the case of Lex we can also decide whether the lexicographic comparison of the arguments is done left-to-right (as in the definition) or right-to-left or in any other fix order.

$\succeq_{rpo}$ defines a congruence on terms $=_{Mul}$ obtained by permuting equivalent arguments below multiset symbols. This *congruence of permutations* is generated by all axioms of the form $f(x_1, \ldots, x_n) = f(x_{\sigma(1)}, \ldots, x_{\sigma(n)})$, for all $f \in Mul$, and all σ in the symmetric group of order n.

3 Higher-Order Rewrite Orderings

The higher-order setting requires generalizing the first-order definition of a reduction ordering:

Definition 3. A quasi-ordering $\geq$ on higher-order terms is *α-compatible* if $u > v$ implies $u' > v'$ for all ground terms in normal form u, v, u', v' such that $u =_\alpha u'$ and $v =_\alpha v'$.

A quasi-ordering $\geq$ on higher-order terms is *monotonic*, also called *stable under ground contexts*, if $u > v$ implies $s[u]_p > s[v]_p$ for all terms u, v, s and positions p for which $s[u]_p$ and $s[v]_p$ are ground terms in normal form.

A quasi-ordering $\geq$ on higher-order terms is *stable under substitutions* if $u > v$ implies $(u\gamma)\downarrow > (v\gamma)\downarrow$ for all terms u, v in normal form and substitutions γ.

A *higher-order rewrite ordering* is an α-compatible, monotonic, stable under substitutions quasi-ordering on higher-order terms. A *higher-order reduction ordering* is a higher-order rewrite ordering which is well-founded on ground terms in normal form.

Lemma 4. *Let R be a set of higher-order rewrite rules. Then R is terminating on terms in normal form iff there exists a higher-order reduction ordering $\geq$ such that $l > r$ for each rewrite rule $l \to r \in R$.*

Proof. The only if case is obtained by taking the α-compatible closure of $\longrightarrow_R^*$ as the quasi-ordering. For the if case, we show as in the first-order case that $s \longrightarrow_R t$ implies $s > t$. As usual, we can assume that all terms in the derivation are ground. By definition $s|_p = (l'\gamma)\downarrow$ for some copy $l' \to r'$ of $l \to r \in R$, position $p \in \mathcal{D}om(s)$ and substitution γ, since s is in normal form. Then, $t = s[(r'\gamma)\downarrow]_p$, which is in normal form by Corollary 2. Now, $l > r$ by assumption, $l' > r'$ by α-compatibility of $>$, $(l\gamma)\downarrow > (r\gamma)\downarrow$ by stability under substitutions of $>$ and $s > t$ by monotonicity of $>$.

Note that, like in the first-order case, the aforementioned properties can be restricted to a subset of the set of all higher-order terms, then the lemma ensures the termination of the rewrite system over this subset. In fact, this is the way we are going to use this result, since we will provide a higher-order reduction ordering on, what we call, *type compatible terms*, which ensures us termination over this set of terms. Afterwards we will study the general case in a different way.

4 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 on type compatible terms 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, which defines a higher-order reduction ordering on type compatible terms. In this second part, the main difficulty is to prove that the induced ordering is closed under appropriate instantiation.

4.1 Typed Recursive Path Ordering

Given a well-founded ordering on types, any reduction ordering on terms can be easily transformed into a reduction ordering on typed terms by comparing lexicographically their types first and then the terms themselves in the original ordering. We go beyond this first idea, by using the recursive path ordering as our starting ordering on terms. This allows us to apply the same idea recursively when comparing subterms.

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* $\geq_{\mathcal{T}_S}$ is assumed to be well-founded.

Definition 5. An algebraic typed term $s : \sigma$ is *type compatible* if for all its subterms $t : \tau$ we have that $t : \tau$ is type compatible and $\sigma \geq_{\mathcal{T}_S} \tau$.

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. The function symbols enjoy a multiset or lexicographic status. We will also allow right-to-left lexicographic status for our examples.

As usual, a free variable $x : \sigma$ is considered as a status-free function symbol which is not comparable to any other function symbol.

We can now define the typed recursive path ordering:

Definition 6. Let $s : \sigma$ and $t : \tau$ be two algebraic typed terms. Then
$s : \sigma = F(\overline{s}) \succeq t : \tau = G(\overline{t})$ 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 $\overline{s} \succeq_{mul} \overline{t}$, or
(d) $F = G \in Lex$ and $\overline{s} \succeq_{lex} \overline{t}$ and $s \succ t_i$ for all i

where $s \succ t$ iff $s \succeq t$ and $t \not\succeq s$.

Note that 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. Note also that it is impossible to prove that $X(a) \succ X(b)$ in case $a \succ b$. Again, this will be crucial in the next subsection, and we will come back on this second question later. Finally, it is clear (and well known for $\succ_{rpo}$), that, in case 2d, it is enough to check $s \succ t_i$ for those t_i which have not been reached in the comparison $\overline{s} \succeq_{lex} \overline{t}$. We will freely use this remark in our examples.

From now on, in this section, we will consider only type compatible terms, and it will be only made explicit in the main properties. The proofs below are similar to the usual proofs for the recursive path ordering. There are however differences aiming at obtaining shorter proofs.

Lemma 7. *Assume that $s \succeq t$ and $t = g(t_1, \ldots, t_n)$. Then, $\forall i \in [1..n]$, $s \succ t_i$.*

Proof. Let $s : \sigma, t : \tau$ and $t_i : \tau_i$. By definition of $\succeq$, $\sigma \geq_{\mathcal{T}_\mathcal{S}} \tau$ and by compatibility property, $\tau \geq_{\mathcal{T}_\mathcal{S}} \tau_i$ for all $i \in [1..n]$. We prove that $s \succ t_i$ for all $i \in [1..n]$ by induction on $|s| + |t|$, and discuss according to the proof that $s \succeq t$.

1. If $\sigma >_{\mathcal{T}_\mathcal{S}} \tau$ or $\tau >_{\mathcal{T}_\mathcal{S}} \tau_i$, then $\sigma >_{\mathcal{T}_\mathcal{S}} \tau_i$ and we are done.
2. If $s \succeq t$ by case 2a, then $s_j \succeq t$ for some s_j, and we conclude by induction that $s_j \succ t_i$, hence $s \succeq t_i$ by case 2a. Assume that $t_i \succeq s$. Then $t_i \succ s_j$ by induction hypothesis again, resulting in a contradiction. Hence $s \succ t_i$.
3. If $s \succeq t$ by cases 2b or 2d, then $s \succ t_i$ by definition of the ordering.
4. If $s \succeq t$ by case 2c, then there must exist some s_j such that $s_j \succeq t_i$. Hence $s \succeq t_i$ by case 2a. Assume now that $t_i \succeq s$. Then $t_i \succ s_j$ by induction hypothesis, resulting in a contradiction. Hence $s \succ t_i$. □

Corollary 8. *Assume that $s \succeq t$ by case 2a or 2b. Then $s \succ t$.*

Proof. We reason by contradiction, assuming that $t \succeq s$, hence $t \succ s_i$ for all immediate subterms s_i of s by Lemma 7. If $s \succeq t$ by case 2a, that is $s_j \succeq t$ for some s_j, we have a contradiction. If $s \succeq t$ by case 2b, then the only possibility to have $t \succeq s$ is by case 2a, hence $t \succ s$ by the previous argument, contradicting $s \succeq t$. □

Lemma 9. *$s \succeq t$ and $t \succeq s$ iff $s =_{Mul} t$.*

Proof. Note first that $s \succeq t$ if $s =_{Mul} t$. The proof of the converse is performed by induction on $|s|$. Note first that s and t must have the same type if $s \succeq t \succeq s$, and that the proofs that $s \succeq t$ and $t \succeq s$ cannot be by case 2a or 2b by Corollary 8. s and t are therefore headed by the same function symbol, and we can easily conclude by induction. □

Corollary 10. $\succeq$ *is reflexive on type compatible terms.*

Corollary 11. $\succ$ *has the (strict) subterm property on type compatible terms.*

Proof. Note that we use here both Corollary 10 and Lemma 7.

Lemma 12. $\succeq$ *is transitive on type compatible terms.*

Proof. Assume that $s = f(\overline{s}) \succeq t = g(\overline{t}) \succeq u = h(\overline{u})$. The proof is performed by induction on $|s|+|t|+|u|$, and by case distinction according to the proof that $s \succeq t$ and $t \succeq u$. Comparisons involving case 1 are straightforward, using transitivity of the type ordering. Others follow the same pattern as in the corresponding proof for rpo. We write (p, q) to indicate that $s \succeq t$ by case p, and $t \succeq u$ by case q.

1. case $(2a, 2)$. Then $s_i \succeq t \succeq u$ for some s_i. By induction hypothesis, $s_i \succeq u$, hence $s \succeq u$ by case 2a.
2. case $(2(\neq 2a),\ 2a)$. Then $t_i \succeq u$. By Lemma 7, $s \succ t_i$, and by induction hypothesis, $s \succeq u$.
3. case (2b or 2c or 2d, 2b). Then $f \geq_{\mathcal{F}} g >_{\mathcal{F}} h$ and $t \succ u_i$ for all u_i. By induction hypothesis $s \succeq u_i$ for all u_i and $s \succeq t$ by case 2b.
4. case (2b, 2c or 2d). Then $t \succ u_i$ for all u_i by Lemma 7. By induction hypothesis, $s \succeq u_i$ for all u_i. Assume that $u_i \succeq s$. Then $u_i \succeq t$ by induction hypothesis, contradicting $t \succ u_i$. Hence $s \succ u_i$ and $s \succeq u$ by case 2b.
5. If both comparisons are by case 2c or by case 2d, we proceed easily by induction. □

Lemma 13. $\succ$ *is stable under ground contexts for type compatible terms of the same type.*

Proof. Assume that s and t are terms of the same type σ such that $s \succ t$, and let $C[\,]_p$ be a ground context with a hole of type σ. We then show easily by induction on the length of p that $C[s] \succ C[t]$. The base case is obtained for $p = \Lambda$, hence is trivial. For the general case, say $p = i \cdot q$ with $i \in \mathbb{N}$, we use first induction with the context $(C|_i)[\,]_q$, and then discuss by case 2d or 2c according to the status of the root of $C[\,]_p$. □

Lemma 14. $\succ$ *is stable under substitutions.*

Proof. We prove that $s \succ t$ implies $s\gamma \succ t\gamma$, for all substitutions γ by induction on $|s| + |t|$. Let $s : \sigma = F(s_1, \ldots, s_n)$ and $t : \tau = G(t_1, \ldots, t_m)$. We distinguish five cases, according to the proof that $s \succ t$:

1. $\sigma >_{\mathcal{T}_S} \tau$. Since types are preserved under substitution, we still have $s\gamma \succ t\gamma$ by comparing their types.
2. $s_i \succeq t$ for some i. By induction hypothesis, $s_i\gamma \succeq t\gamma$, and hence $s\gamma \succ t\gamma$ by case 2a of the definition of the ordering.

3. $F >_{\mathcal{F}} G$ and $s \succ t_i$ for all i. By induction, $s\gamma \succ t_i\gamma$, and therefore $s\gamma \succ t\gamma$ by case 2b of the definition of the ordering.
4. $F = G \in Mul$ and $\overline{s} \succ_{mul} \overline{t}$. By induction, $\overline{s\gamma} \succ_{mul} \overline{t\gamma}$, which proves this case by case 2c of the definition of the ordering.
5. $F = G \in Lex$, $\overline{s} \succ_{lex} \overline{t}$ and $s \succ t_i$ for all i. By induction, $\overline{s\gamma} \succ_{lex} \overline{t\gamma}$ and $s\gamma \succ t_i\gamma$, which proves this case as well. □

We are left with the proof of well-foundedness for type compatible terms, 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 15. *Let $\{s_i\}_{i \in \mathbb{N}}$ be an infinite sequence of ground type compatible 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 with respect to the well-founded type ordering $>_{\mathcal{T}_S}$. Among such sequences of minimal type σ, we choose a minimal one (in the usual sense, by comparing the sizes 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 $\{F(\overline{u^j})\}_{j \in \mathbb{N}}$, all headed by some function symbol $F \in \mathcal{F}_{\sigma_1 \times \ldots \times \sigma_n \to \sigma}$ such that, by compatibility property of the terms in the sequence, $\sigma \geq_{\mathcal{T}_S} \sigma_k$ for all $k \in [1..n]$ and all terms in each $\overline{u^j}$ are type compatible. We now proceed to show that, for each $k \in [1..n]$, infinite sequences of type σ_k built up from the set $IS = \{u^j_k\}_{k \in [1..n], j \in \mathbb{N}}$ of immediate subterms of terms in the extracted sequence 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 then show as usual that n-tuples of type $\sigma_1, \ldots, \sigma_n$ of terms in IS satisfy the lemma with respect to the product ordering generated by the embedding. We can therefore find indices $i < j$ such that for all $k \in [1..n]$, the term u^i_k is embedded in the term u^j_k. Then $F(\overline{u^i})$ is embedded in $F(\overline{u^j})$ contradicting our assumption that there was a counterexample sequence. □

Theorem 16. *$\succeq$ is a well-founded simplification ordering on type compatible ground algebraic terms.*

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.

4.2 Recursive Path Ordering for Normalized Terms

The ordering for normalized terms works by comparing interpretations of normalized terms in the typed recursive path ordering we have just defined. Because bound variables cannot be substituted, they will be considered as new (non-variable) symbols $\perp_\sigma$ of their respective output 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:

$$\begin{array}{rcl} \mathcal{X}_\mathcal{T} & = & \{x_\sigma \mid x : \sigma \in \mathcal{X}\} \\ \mathcal{F}_\mathcal{T} & = & \{F_\sigma \mid F \in \mathcal{F}_{\sigma_1 \times \ldots \times \sigma_n \to \sigma}\} \\ \lambda\mathcal{F} & = & \mathcal{X}_\mathcal{T} \cup \mathcal{F} \cup \bigcup\limits_{\sigma \in \mathcal{T}_\mathcal{S}} \{\perp_\sigma\} \cup \bigcup\limits_{\sigma, \tau \in \mathcal{T}_\mathcal{S}} \{\lambda_{\sigma \to \tau}\} \end{array}$$

Note that we have a single symbol $\perp_\sigma$ for each type σ and a single $\lambda_{\sigma\to\tau}$ operator for each arrow type $\sigma \to \tau$. 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. Note also that we would have more flexibility in practice to deal with arbitrary overloaded function symbols if they were indexed by their input and output types, instead of by their output type only. On the other hand, this would complicate our notations a little bit.

Definition 17. The interpretation function maps normalized terms to typed algebraic terms over the new signature:

$$\begin{array}{rcl} [\![x : \sigma]\!] & = & x_\sigma \\ [\![(\lambda x.u) : \sigma \to \tau]\!] & = & \lambda_{\sigma \to \tau}([\![u]\!]\{x_{O(\sigma)} \mapsto \perp_{O(\sigma)}\}) \\ [\![F(\ldots, u_i, \ldots) : \sigma]\!] & = & F_\sigma(\ldots, [\![u_i]\!], \ldots) \end{array}$$

In the above definition of the interpretation function, the reader should be aware that $X_\tau(a_1, \ldots, a_n)$ denotes the application of the higher-order variable $X : \tau_1 \to \ldots \to \tau_n \to \tau$ (with τ basic) to the expressions $a_i : \tau_i$. Substituting X_τ by the abstraction $\lambda y_1 \ldots y_n . \perp_\tau(y_1, \ldots, y_n)$ and then β-normalizing the result, yields the term $\perp_\tau(a_1, \ldots, a_n)$, as if X_τ had been simply replaced by $\perp_\tau$. This justifies the shortcut used above, in which we substitute first-order variables and higher-order variables by the corresponding constant or funtion symbol $\perp_\tau$.

We define type compatibility for higher-order terms in the same way as we did for algebraic terms, i.e. $s : \sigma$ is type compatible if for every position $p \neq \Lambda$ in s we have that $s|_p : \tau$ is type compatible and $\sigma \geq_{\mathcal{T}_\mathcal{S}} \tau$. Then it holds that s is type compatible iff $[\![s]\!]$ is type compatible.

Before we can introduce our ordering, we need to define the precedence on the new signature. It will simply extend the precedence $>_{\mathcal{F}}$ over the former signature to $>_{\lambda\mathcal{F}}$, by adding new pairs for taking care of the $\perp_\sigma$ and $\lambda_{\sigma\to\tau}$ operators. In practice, it is wise to have $\perp_\sigma$ minimal among the function symbols of output type σ. 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 5.

Definition 18. Assuming that s and t are two typable normalized terms such that no variable is both free and bound nor bound twice in the pair (s,t), we define:

$$s \succeq_{horpo} t \text{ iff } [\![s]\!] \succeq [\![t]\!]$$

It is straightforward to see that the strict part of $\succeq_{horpo}$ fulfils $s \succ_{horpo} t$ iff $[\![s]\!] \succ [\![t]\!]$. Similarly, by Lemma 9, we have that $s \succeq_{horpo} t$ and $t \succeq_{horpo} s$ iff $[\![s]\!] =_{Mul} [\![t]\!]$.

Lemma 19. *Let x and y be two variables of the same type and u a term such that $x \notin \mathcal{BV}ar(u)$, $y \notin \mathcal{BV}ar(u)$ and σ is the output type of x. Then $[\![u\{x \mapsto y\}]\!] = [\![u]\!]\{x_\sigma \mapsto y_\sigma\}$.*

Proof. By induction on $|u|$ and case analysis on the definition of the interpretation:

1. $u = x$. Then $[\![u\{x \mapsto y\}]\!] = [\![y]\!] = y_\sigma$ and $[\![u]\!]\{x_\sigma \mapsto y_\sigma\} = x_\sigma\{x_\sigma \mapsto y_\sigma\} = y_\sigma$.
2. $u = z : \tau$ for some variable z different from x. Then $[\![u\{x \mapsto y\}]\!] = [\![z]\!] = z_\tau$ and $[\![u]\!]\{x_\sigma \mapsto y_\sigma\} = z_\tau\{x_\sigma \mapsto y_\sigma\} = z_\tau$.
3. $u = (\lambda z.v) : \tau_1 \to \tau_2$. Then $[\![u\{x \mapsto y\}]\!] = [\![\lambda z.(v\{x \mapsto y\})]\!] = \lambda_{\tau_1\to\tau_2}([\![v\{x \mapsto y\}]\!]\{z_{O(\tau_1)} \mapsto \perp_{O(\tau_1)}\})$ and by induction hypothesis it is equal to $\lambda_{\tau_1\to\tau_2}([\![v]\!]\{x_\sigma \mapsto y_\sigma\}\{z_{O(\tau_1)} \mapsto \perp_{O(\tau_1)}\}) = \lambda_{\tau_1\to\tau_2}([\![v]\!]\{z_{O(\tau_1)} \mapsto \perp_{O(\tau_1)}\})\{x_\sigma \mapsto y_\sigma\}$, which is equal to $[\![u]\!]\{x_\sigma \mapsto y_\sigma\}$.
4. $u = x(\ldots, u_i, \ldots) : \sigma$. Then $[\![u\{x \mapsto y\}]\!] = [\![y(\ldots, u_i\{x \mapsto y\}, \ldots)]\!] = y_\sigma(\ldots, [\![u_i\{x \mapsto y\}]\!], \ldots)$, which by induction hypothesis is equal to $y_\sigma(\ldots, [\![u_i]\!]\{x_\sigma \mapsto y_\sigma\}, \ldots) = y_\sigma(\ldots, [\![u_i]\!], \ldots)\{x_\sigma \mapsto y_\sigma\} = (x_\sigma(\ldots, [\![u_i]\!], \ldots))\{x_\sigma \mapsto y_\sigma\}$, which is equal to $[\![u]\!]\{x_\sigma \mapsto y_\sigma\}$.
5. $u = F(\ldots, u_i, \ldots) : \sigma$, for some F different form x. Then we proceed as in the previous case. □

Lemma 20. *If $u =_\alpha v$ then $[\![u]\!] = [\![v]\!]$, for all terms u and v.*

Proof. We will prove that $u =_\alpha u'$ in one step implies $[\![u]\!] = [\![u']\!]$, then we can conclude by induction on the number of steps in the derivation of $u =_\alpha v$.

We proceed by induction on $|u|$:

1. If $u = \lambda x.w : \sigma \to \tau$ and $u' = \lambda y.w\{x \mapsto y\}$, with $y \notin \mathcal{BV}ar(w) \cup \mathcal{V}ar(w)$, then we have $[\![u]\!] = \lambda_{\sigma\to\tau}([\![w]\!]\{x_{O(\sigma)} \mapsto \perp_{O(\sigma)}\})$ and $[\![u']\!] = \lambda_{\sigma\to\tau}([\![w\{x \mapsto y\}]\!]\{y_{O(\sigma)} \mapsto \perp_{O(\sigma)}\})$, now by Lemma 19 we have $[\![u']\!] = \lambda_{\sigma\to\tau}([\![w]\!]\{x_{O(\sigma)} \mapsto y_{O(\sigma)}\}\{y_{O(\sigma)} \mapsto \perp_{O(\sigma)}\})$, which is equal to $[\![u]\!]$.

2. If $u = \lambda x.w : \sigma \to \tau$ and $u' = \lambda x.w' : \sigma \to \tau$, with $w =_\alpha w'$ in one step, then by induction hypothesis $[\![w]\!] = [\![w']\!]$. Therefore, $[\![u]\!] = \lambda_{\sigma\to\tau}([\![w]\!]\{x_{O(\sigma)} \mapsto \perp_{O(\sigma)}\}) = \lambda_{\sigma\to\tau}([\![w']\!]\{x_{O(\sigma)} \mapsto \perp_{O(\sigma)}\}) = [\![u']\!]$.
3. If $u = F(\ldots u_i \ldots) : \sigma$ and $u' = F(\ldots u'_i \ldots) : \sigma$, with $u_i =_\alpha u'_i$ for some i, then by induction hypothesis $[\![u_i]\!] = [\![u'_i]\!]$. Therefore $[\![u]\!] = F_\sigma(\ldots [\![u_i]\!] \ldots) = F_\sigma(\ldots [\![u'_i]\!] \ldots) = [\![u']\!]$. □

Corollary 21. *$\succeq_{horpo}$ is α-compatible (with α-equality).*

Lemma 22. *s is a ground normalized term iff $[\![s]\!]$ is a ground algebraic term.*

Proof. This is shown by means of a more general result, namely $x : \sigma \in \mathcal{V}ar(s)$ iff $x_{O(\sigma)} \in \mathcal{V}ar([\![s]\!])$, by induction on $|s|$ and case analysis on the definition of the interpretation in a similarly way as in the proof of previous lemmas.

Lemma 23. *$\succeq_{horpo}$ is a well-founded quasi-ordering on type compatible ground normalized terms.*

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

Lemma 24. *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 typable ground normalized terms.*

Proof. We will prove that $[\![u[s]]\!] = [\![u]\!][[\![s]\!]\gamma]$ and $[\![u[t]]\!] = [\![u]\!][[\![t]\!]\gamma]$ for some substitution γ taking its value in $\{\perp_\sigma \mid \sigma \in \mathcal{T}_\mathcal{S} \wedge \sigma \text{ is basic}\}$. Note that we are considering that the interpretation of a hole is a hole, and hence $[\![u[\,]]\!] = [\![u]\!][\,]$. Then we can conclude that $[\![u[s]]\!] \succ [\![u[t]]\!]$: since $[\![s]\!] \succ [\![t]\!]$ holds by assumption that $s \succ_{horpo} t$, we have $[\![s]\!]\gamma \succ [\![t]\!]\gamma$ by stability of $\succ$; finally as, by Lemma 22 the interpretation preserves groundness, $[\![u]\!]$ is a ground context and hence by monotonicity of $\succ$ under ground context we have $[\![u[s]]\!] = [\![u]\!][[\![s]\!]\gamma] \succ [\![u]\!][[\![t]\!]\gamma] = [\![u[t]]\!]$.

To prove $[\![u[s]]\!] = [\![u]\!][[\![s]\!]\gamma]$ and $[\![u[t]]\!] = [\![u]\!][[\![t]\!]\gamma]$, we proceed by induction on $|u|$.

If the context is empty it trivially holds by taking γ as the empty substitution. Otherwise, since the context is ground, two cases have to be considered:

1. If $u[\,] = F(\ldots, u'[\,], \ldots)$ for some algebraic function symbol F with output type σ, then we have $[\![u[s]]\!] = F_\sigma(\ldots, [\![u'[s]]\!], \ldots)$ and $[\![u[t]]\!] = F_\sigma(\ldots, [\![u'[t]]\!], \ldots)$. By induction hypothesis we have $[\![u[s]]\!] = F_\sigma(\ldots, [\![u']\!][[\![s]\!]\gamma], \ldots)$ and $[\![u[t]]\!] = F_\sigma(\ldots, [\![u']\!][[\![t]\!]\gamma], \ldots)$ for some γ, and therefore $[\![u[s]]\!] = [\![u]\!][[\![t]\!]\gamma]$ and $[\![u[t]]\!] = [\![u]\!][[\![t]\!]\gamma]$.
2. Otherwise, let $u[\,]$ be $\lambda x.u'[\,]$ with type $\sigma \to \tau$. Then $[\![u[s]]\!] = \lambda_{\sigma\to\tau}([\![u'[s]]\!]\{x_{O(\sigma)} \mapsto \perp_{O(\sigma)}\})$ and $[\![u[t]]\!] = \lambda_{\sigma\to\tau}([\![u'[t]]\!]\{x_{O(\sigma)} \mapsto \perp_{O(\sigma)}\})$. By induction hypothesis $[\![u[s]]\!] = \lambda_{\sigma\to\tau}([\![u']\!][[\![s]\!]\gamma']\{x_{O(\sigma)} \mapsto \perp_{O(\sigma)}\})$ and $[\![u[t]]\!] = \lambda_{\sigma\to\tau}([\![u']\!][[\![t]\!]\gamma']\{x_{O(\sigma)} \mapsto \perp_{O(\sigma)}\})$. Now, taking $\gamma = \gamma' \cup \{x_{O(\sigma)} \mapsto \perp_{O(\sigma)}\}$, we have $[\![u[s]]\!] = \lambda_{\sigma\to\tau}([\![u']\!]\{x_{O(\sigma)} \mapsto \perp_{O(\sigma)}\}[[\![s]\!]\gamma]) = [\![u]\!][[\![s]\!]\gamma]$ and $[\![u[t]]\!] = \lambda_{\sigma\to\tau}([\![u']\!]\{x_{O(\sigma)} \mapsto \perp_{O(\sigma)}\}[[\![t]\!]\gamma]) = [\![u]\!][[\![t]\!]\gamma]$. □

Lemma 25. $\succ_{horpo}$ *is stable under 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 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 $\bot_\sigma$) 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 $\{\bot_\sigma \mid \sigma \in \mathcal{T_S} \wedge \sigma \text{ is basic}\}$, and such that if $X_{\sigma'} \in \mathcal{D}om(\psi_1) \cup \mathcal{D}om(\psi_2)$ (for some type σ') then $X \notin \mathcal{D}om(\gamma)$ and $X \notin \mathcal{V}ar(y\gamma)$ for all $y \in \mathcal{D}om(\gamma)$. 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 $F_\sigma\psi_1 = \bot_\sigma$; or else (ii) s is an abstraction $\lambda x.s_1$, and the output type of x is σ'.
   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, $s'_i\psi_1 \succ t'\psi_2$ 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 $s'_1 = [\![s_1]\!]\{x_{\sigma'} \mapsto \bot_{\sigma'}\}$. Then $s'_1\psi_1$ can be written as $[\![s_1]\!]\psi'_1$, with $\psi'_1 = \{x_{\sigma'} \mapsto \bot_{\sigma'}\} \cup \psi_1$. Note that x cannot appear neither in the domain of γ nor in $y\gamma$ for all y in its domain 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 $[\![(\lambda x.s_1)\gamma\downarrow]\!]\psi_1 = \lambda_\sigma(([\![s_1\gamma\!\downarrow]\!]\{x_{\sigma'} \mapsto \bot_{\sigma'}\})\psi_1) = \lambda_\sigma([\![s_1\gamma\!\downarrow]\!]\psi'_1)$.
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 $t'_1 = [\![t_1]\!]\{x_{\sigma'} \mapsto \bot_{\sigma'}\}$ for some $x : \sigma'$. Since $t'_1\psi_2$ can be written as $[\![t_1]\!]\psi'_2$ (with $\psi'_2 = \{x_{\sigma'} \mapsto \bot_{\sigma'}\} \cup \psi_2$), 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 $s'_1 = [\![s_1]\!]\{x_{\sigma'} \mapsto \bot_{\sigma'}\}$ for some $x : \sigma'$, and $t'\psi_2 = \lambda_\sigma(t'_1\psi_2)$, where $t'_1 = [\![t_1]\!]\{y_{\sigma'} \mapsto \bot_{\sigma'}\}$ for some $y : \sigma'$. Since $s'_1\psi_1$ can be written as $[\![s_1]\!]\psi'_1$ (with $\psi'_1 = \{x_{\sigma'} \mapsto \bot_{\sigma'}\} \cup \psi_1$) and $t'_1\psi_2$ can be written as $[\![t_1]\!]\psi'_2$ (with $\psi'_2 = \{y_{\sigma'} \mapsto \bot_{\sigma'}\} \cup \psi_2$), 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 $\overline{s'\psi_1} \succ_{mul} \overline{t'\psi_2}$. By induction, the result holds.
5. $F_\sigma\psi_1 = G_\tau\psi_2 \in Lex$ and $\overline{s'\psi_1} \succ_{lex} \overline{t'\psi_2}$ 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 26. *$\succeq_{horpo}$ is a higher-order reduction ordering on type compatible terms.*

By Lemma 4 this ensures us that the ordeing can be used to prove termination of a rewriting system on type compatible terms.

4.3 Totality

Lemma 27. *Assume that the type ordering $\geq_{\mathcal{T}_S}$ is total and the precedence $\geq_{\lambda\mathcal{F}}$ is total on function symbols of the signature $\mathcal{F}$ which have the same output type. Then $\succeq$ is a total ordering on ground type compatible algebraic terms.*

Proof. Assume $s = f(\overline{s}), t = g(\overline{t})$ is the minimal pair of ground algebraic terms such that s and t are incomparable. Then s and t must have the same type. By minimality assumption, the pairs $(s, t_j), (s_i, t), (s_i, t_j)$ are comparable for all $s_i \in \overline{s}$ and $t_j \in \overline{t}$. If $t_j \succeq s$ for some $t_j \in \overline{t}$, then $t \succ s$, a contradiction. Hence $s \succ t_j$ for all $t_j \in \overline{t}$. By argument of symmetry, $t \succ s_i$ for all $s_i \in \overline{s}$. There are now several cases:

1. $f >_{\mathcal{F}} g$. Then $t \succ s$ by case 2b, a contradiction.
2. $g >_{\mathcal{F}} f$ is a symmetric case.
3. $f = g \in Mul$. Since all pairs (s_i, t_j) are comparable, either $\overline{s} \succeq_{mul} \overline{t}$, in which case $s \succeq t$, or $\overline{t} \succeq_{mul} \overline{s}$, in which case $t \succeq s$, yielding a contradiction in both cases.
4. $f = g \in Lex$. Again, either $\overline{s} \succeq_{lex} \overline{t}$, in which $s \succeq t$ since $s \succ t_j$ for all $t_j \in \overline{t}$, or $\overline{t} \succeq_{lex} \overline{s}$, in which $t \succeq s$ since $t \succ s_i$ for all $s_i \in \overline{s}$, yielding a contradiction in both cases again. □

As a consequence of Lemma 27 and Lemma 9, the strict part of the ordering orients any two ground algebraic terms which are not in the same equivalence class for $=_{Mul}$, when the type ordering $\geq_{\mathcal{T}_S}$ is total and the precedence $\geq_{\lambda\mathcal{F}}$ is total on function symbols of the extended signature $\lambda\mathcal{F}$ which have the same output type.

Lemma 28. *$\succeq_{horpo}$ is total on ground terms when the type ordering $\geq_{\mathcal{T}_S}$ is total and the precedence $\geq_{\lambda\mathcal{F}}$ is total on function symbols of the extended signature $\lambda\mathcal{F}$ which have the same output type.*

5 Compatible Terms and Consistent Type Orderings

The goal of this section is to provide sufficient conditions ensuring the compatibility property of the terms wrt. the type ordering. We will also show that for some non-compatible terms the strong normalization is also assured by the termination of the rewrite system on compatible terms.

Definition 29. A type σ is compatible with the quasi-ordering on types $\geq_{\mathcal{T}_S}$ iff it is a sort or it is of the form $\sigma_1 \to \ldots \to \sigma_n \to \tau$, where τ is a sort, and $\tau \geq_{\mathcal{T}_S} \sigma_i$ and σ_i is compatible for every $i \in \{1 \ldots n\}$. Analogously we can define compatibility of a type declaration.

Definition 30. The quasi-ordering on types $\geq_{\mathcal{T}_S}$ is *consistent with the type structure* if
(i) $\forall f : \sigma \in \mathcal{F}$, σ is a compatible type,
(ii) $\sigma \to \tau \geq_{\mathcal{T}_S} \tau$.

Lemma 31. *Let $\geq_{\mathcal{T}_S}$ be a quasi-ordering on types consistent with the type structure and let $s : \sigma$ be a normalized term. Then $s : \sigma$ is a compatible term if*

(i) for every $x : \tau \in \mathcal{V}ar(s)$ we have that τ is a compatible type, and
(ii) if $s : \sigma$ is of the form $\lambda x_1 \ldots x_n.u : \sigma_1 \to \ldots \to \sigma_n \to \rho$, for some basic type ρ, then σ_i is a compatible type for all $x_i \in \mathcal{V}ar(u)$.

Proof. We proceed by induction on $|s|$. The property trivially holds if s is a constant or a basic type variable. Otherwise, there are three possible cases:

1. $s = X(s_1, \ldots, s_n)$, with $X \in \mathcal{V}ar(s)$ and $s_i : \sigma_i$. By condition (i), $\sigma_1 \to \ldots \to \sigma_n \to \sigma$ is a compatible type, and hence $\sigma \geq_{\mathcal{T}_S} \sigma_i$ and σ_i is a compatible type for every i. Then, since $\mathcal{V}ar(s_i) \subseteq \mathcal{V}ar(s)$, $s_i : \sigma_i$ satisfies condition (i), and since σ_i is a compatible type, it also fulfils condition (ii). Therefore, by induction hypothesis, $s_i : \sigma_i$ is a compatible term, and, since $\sigma \geq_{\mathcal{T}_S} \sigma_i$ for every i, we have that $s : \sigma$ is a compatible term.
2. $s = F(s_1, \ldots, s_n)$, with $s_i : \sigma_i$. By consistency of the type ordering with the type structure we have $\sigma \geq_{\mathcal{T}_S} \sigma_i$ and σ_i is a compatible type for every i. Now we conclude as in the previous case.
3. $s = \lambda x_1 \ldots x_n.u$, with $n \geq 1$, $\sigma = \sigma_1 \to \alpha$ and $\alpha = \sigma_2 \to \ldots \to \sigma_n \to \rho$. Then, by (ii) if $x \in \mathcal{V}ar(u)$ then σ_1 is a compatible type, which implies that $\lambda x_2 \ldots x_n.u : \alpha$ fulfils (i). On the other hand, since $\lambda x_1 \ldots x_n.u : \sigma_1 \to \ldots \to \sigma_n \to \rho$ fulfils condition (ii), we have that $\lambda x_2 \ldots x_n.u : \sigma_2 \to \ldots \to \sigma_n \to \rho$, fulfils condition (ii) too. Therefore, by induction hypothesis $\lambda x_2 \ldots x_n.u : \alpha$ is a compatible term, and hence, since, by consistency of $\geq_{\mathcal{T}_S}$ with the type structure, $\sigma \geq_{\mathcal{T}_S} \alpha$, we have that $s : \sigma$ is a compatible term. □

These restrictions on the terms are indeed necessary to ensure well-foundedness as shown in the following example:

Example 1. Consider the following signature
$\mathcal{S} = \{\alpha, \beta\}$,
$\mathcal{F}_{\alpha\to\beta} = \{h\}, \mathcal{F}_{(\alpha\to\alpha)\to\alpha} = \{g\}, \mathcal{F}_{\alpha\times\alpha\to\beta} = \{f\}$,
$\mathcal{X}_{\alpha} = \{y\}, \mathcal{X}_{\alpha\to\alpha} = \{X\}$

and the rule:

$$f(g(\lambda x.X(x)), y) \;\to\; h(X(y))$$

This rule can be proved terminating by our ordering (see next section), using a quasi-ordering on types consistent with the type structure in which β is strictly greater than α. Now consider the non-compatible free variable $Z : \beta \to \alpha$, then the term $f(g(\lambda x.Z(f(x,x))), g(\lambda x.Z(f(x,x))))$ can be rewritten to $h(Z(f(g(\lambda x.Z(f(x,x))), g(\lambda x.Z(f(x,x))))))$ which includes the initial redex again.

Of course if we add a binder for Z making the term ground, the resulting term would have a bound variable Z with a non-compatible type, and hence Lemma 31 cannot be applied either.

Lemma 32. *Let $\geq_{\mathcal{T}_S}$ be a quasi-ordering on types which is consistent with the type structure and let R be a higher-order term rewriting system terminating on compatible terms and such that all rules have a compatible type. Then all ground normalized terms $t : \sigma_1 \to \ldots \to \sigma_n \to \sigma$, with σ basic and σ_i compatible for all $i \in \{1\ldots n\}$, are strongly normalizable.*

Proof. By Lemma 1 t is of the form $\lambda x_1 \ldots x_m.F(u_1, \ldots, u_n)$, for some $m \geq 0$ and $F \in \mathcal{F}\cup\mathcal{X}$ such that $F(u_1, \ldots, u_n) : \sigma$ and $x_i : \sigma_i$, for every $i \in \{1\ldots n\}$. Let t' be $\lambda x_j \ldots x_m.F(u_1, \ldots, u_n)$ the biggest subterm of t which is type compatible (note that at least $F(u_1, \ldots, u_n)$ is type compatible). Then by Lemma 31 t' is a compatible term, since, by assumption, its free variables, included in $\{x_1, \ldots, x_{j-1}\}$, as well as its remaining bound variables in the head, i.e. $\{x_j, \ldots, x_m\}$, have a compatible type. Now, since all rules in R have a compatible type, it is obvious that if there is an infinite sequence starting with t there must be one starting with t', which will contradict the termination of R on compatible terms. □

This lemma means that by using $\succ_{horpo}$, with an underlying ordering on types fulfiling the consistency restrictions, we can ensure strong normalization of every ground normalized term except the ones that are headed by some lambdas and at least one of them binds a variable with non-compatible type. Note that this requirement is strictly weaker than requiring the terms to have a compatible type. In fact, as we have seen in Lemma 31, we can slightly strength the previous lemma, by requiring compatibility on the bound variables that occur at least once in the body of the term.

6 Type Orderings and Examples

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

6.1 Recursive Path Ordering on Types

As already said, the recursive path ordering itself is a good candidate for the type ordering. Condition (ii) of consistency with the type structure 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 $\rightarrow$ smaller than any basic type, although there is no obligation. The only requirement is to provide a precedence $\geq_{\mathcal{S}}$ on $\mathcal{S} \cup \{\rightarrow\}$, and a status for $\rightarrow$. The obtained ordering is quite powerful, hence it will allow us to compare many terms by type.

Example 1: Sorting (see Section 2.6)
We use the following precedences:

$List >_{\mathcal{S}} Nat, List >_{\mathcal{S}} \rightarrow$
$\{max, min\} >_{\lambda\mathcal{F}} \{0, s\}$,
$insert >_{\lambda\mathcal{F}} cons, \; sort >_{\lambda\mathcal{F}} \{nil, insert\}$,
$\{ascending\_sort, descending\_sort\} >_{\lambda\mathcal{F}} sort$,
$insert \in Lex$ (right to left) and $sort \in Mul$.

We will check only two rules:

$$sort(cons(n, l), \lambda xy.X(x, y), \lambda xy.Y(x, y)) \succ_{horpo}$$
$$insert(n, sort(l, \lambda xy.X(x, y), \lambda xy.Y(x, y)), \lambda xy.X(x, y), \lambda xy.Y(x, y))$$

and

$$insert(n, cons(m, l), \lambda xy.X(x, y), \lambda xy.Y(x, y)) \succ_{horpo}$$
$$cons(X(n, m), insert(Y(n, m), l, \lambda xy.X(x, y), \lambda xy.Y(x, y)))$$

leaving the others to the reader.

For the first rule, we first apply the interpretation before applying the ordering on algebraic terms (for readability reasons we will use N for Nat and L for $List$), hence we need to prove that

$$s = sort_L(cons_L(n_N, l_L), \lambda_{N\rightarrow N}(X_N(\bot_N, \bot_N)), \lambda_{N\rightarrow N}(Y_N(\bot_N, \bot_N))) \succ$$
$$insert_L(n_N, sort_L(l_L, \lambda_{N\rightarrow N}(X_N(\bot_N, \bot_N)), \lambda_{N\rightarrow N}(Y_N(\bot_N, \bot_N))),$$
$$\lambda_{N\rightarrow N}(X_N(\bot_N, \bot_N)), \lambda_{N\rightarrow N}(Y_N(\bot_N, \bot_N)))$$

since $sort >_{\lambda\mathcal{F}} insert$, we need to check recursively

$s \succ n_N$,
$s \succ sort_L(l_L, \lambda_{N\rightarrow N}(X_N(\bot_N, \bot_N)), \lambda_{N\rightarrow N}(Y_N(\bot_N, \bot_N)))$,
$s \succ \lambda_{N\rightarrow N}(X_N(\bot_N, \bot_N))$,

and

$s \succ \lambda_{N\rightarrow N}(Y_N(\bot_N, \bot_N))$.

The second comparison succeeds by case 2c, using the fact that $cons_L(n_N, l_L) \succ l_L$, and all other cases by case 1.

For the second rule, after applying the interpretation, since $insert >_{\lambda\mathcal{F}} cons$, we need to check recursively $t =$

$insert_L(n_N, cons_L(m_N, l_L), \lambda_{N\to N}(X_N(\perp_N, \perp_N)), \lambda_{N\to N}(Y_N(\perp_N, \perp_N))) \succ X_N(n_N, m_N)$,

and

$t \succ insert_L(Y_N(n_N, m_N), l_L, \lambda_{N\to N}(X_N(\perp_N, \perp_N)), \lambda_{N\to N}(Y_N(\perp_N, \perp_N)))$

The first comparison is by case 1; for the second, since $insert$ has a right-to-left lexicographic status, the comparison $cons_L(m_N, l_L) \succ l_L$ succeeds, and we are left with

$t \succ Y_N(n_N, m_N)$,

which works by case 1 again.

Example 2: (Example 1 in the previous section)
We use the following precedences to prove the termination:

$$\beta >_{\mathcal{S}} \alpha, \quad \beta >_{\mathcal{S}} \to, \qquad f >_{\lambda\mathcal{F}} h.$$

Example 3: Mapcar
$\mathcal{S} = \{Nat, List\}$,
$\mathcal{F}_{List} = \{nil\}, \mathcal{F}_{Nat\times List\to List} = \{cons\}, \mathcal{F}_{(Nat\to Nat)\times List\to List} = \{map\}$,
$\mathcal{X}_{Nat\to Nat} = \{X\}, \mathcal{X}_{List} = \{l\}, \mathcal{X}_{Nat} = \{n\}$

$$\begin{array}{rcl} map(\lambda x.X(x), nil) & \to & nil \\ map(\lambda x.X(x), cons(n, l)) & \to & cons(X(n), map(\lambda x.X(x), l)) \end{array}$$

The termination proof may use the following precedences:

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

Example 4: Maplist
$\mathcal{S} = \{Nat, List\}$,
$\mathcal{F}_{List} = \{nil\}, \mathcal{F}_{Nat\times List\to List} = \{cons\}, \mathcal{F}_{(Nat\to Nat)\times List\to List} = \{fcons\}$,
$\mathcal{F}_{List\times Nat\to List} = \{fmap\}$,
$\mathcal{X}_{Nat\to Nat} = \{X\}, \mathcal{X}_{List} = \{L\}, \mathcal{X}_{Nat} = \{n\}$

$$\begin{array}{rcl} fmap(nil, n) & \to & nil \\ fmap(fcons(\lambda x.X(x), L), n) & \to & cons(X(n), fmap(L, n)) \end{array}$$

The termination proof may use the following precedences:

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

6.2 Sort Orderings

The other two orderings are again generated by a precedence $\geq_{\mathcal{S}}$ on the set of basic types. The first one compares type expressions according to the maximal basic type occurring in them, that is

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

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

As we take the maximal sort occurring in the type expression, this type ordering always fulfils condition (ii) of the consistency with the type structure, and hence, like for the previous type ordering, we just have to check condition (i).

Example 5: Prenex Normal Forms

This example is adapted from [16], where its local confluence is proved via the computation of its (higher-order) critical pairs.

Formulas will be represented as λ-terms with basic type ϕ. The idea is that quantifiers bind variables via the use of a functional argument, that is, an abstraction:
$\mathcal{S} = \{\phi\}$,
$\mathcal{F}_{\phi\to\phi} = \{\neg\}, \mathcal{F}_{\phi\times\phi\to\phi} = \{\wedge, \vee\}, \mathcal{F}_{(\phi\to\phi)\to\phi} = \{\exists, \forall\}$,
$\mathcal{X}_\phi = \{X\}, \mathcal{X}_{\phi\to\phi} = \{Y\}$.

$$\begin{array}{rcl}
X \wedge \forall(\lambda x.Y(x)) & \to & \forall(\lambda x.(X \wedge Y(x))) \\
X \vee \forall(\lambda x.Y(x)) & \to & \forall(\lambda x.(X \vee Y(x))) \\
X \wedge \exists(\lambda x.Y(x)) & \to & \exists(\lambda x.(X \wedge Y(x))) \\
X \vee \exists(\lambda x.Y(x)) & \to & \exists(\lambda x.(X \vee Y(x))) \\
\neg(\forall(\lambda x.Y(x))) & \to & \exists(\lambda x.\neg(Y(x))) \\
\neg(\exists(\lambda x.Y(x))) & \to & \forall(\lambda x.\neg(Y(x)))
\end{array}$$

The missing rules are obtained by symmetry. The termination proof may use the following precedences:
$\wedge >_{\lambda\mathcal{F}} \forall$, $\wedge >_{\lambda\mathcal{F}} \exists$, $\wedge >_{\lambda\mathcal{F}} \lambda_{\phi\to\phi}$, $\vee >_{\lambda\mathcal{F}} \forall$, $\vee >_{\lambda\mathcal{F}} \exists$,
$\vee >_{\lambda\mathcal{F}} \lambda_{\phi\to\phi}$, $\neg >_{\lambda\mathcal{F}} \forall$, $\neg >_{\lambda\mathcal{F}} \exists$, $\neg >_{\lambda\mathcal{F}} \lambda_{\phi\to\phi}$,
$\wedge, \vee \in Mul$.

Note that since $\phi \to \phi$ is necessarily strictly bigger than ϕ for a recursive path ordering, $\forall$ and $\exists$ would not fulfil the consistency property with respect to any recursive path ordering on types.

Finally, the third ordering on types, which can also be used in the previous example, is the weakest one. In this case, given a precedence $\geq_{\mathcal{S}}$ on the set of basic types, we compare type expressions according to their output sort. That is

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

Again, as we take the output type of the type expression, condition (ii) of the consistency with the type structure is always fulfiled.

7 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 [14], 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 [4] 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, [13] describes a computable ordering in the same spirit as ours, based on a first-order interpretation of terms in η-long β-normal form, but a counterexample to its well-foundedness is given in [14]. [12] introduces a powerful notion of higher-order subterm[3], allowing to state that $v(x, u)$ is a subterm of $rec(S(x), u, \lambda yz.v(y, z))$. 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 implicitly consider $\lambda yz.v(y, z)$ as a subterm of $rec(x, u, \lambda yz.v(y, z))$. Applying $\lambda yz.v(y, z)$ to inputs of the appropriate type results in a term of type γ, and compatibility becomes satisfied. This is actually the essence of Loria-Saenz' definition of higher-order subterm [12]. Using this idea should result 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 6. 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 [13, 12], which fail to check the second *insert* rule.

Finally, we can come back on the question of variables, and the example that $X(a)$ and $X(b)$ are incomparable even if a and b are comparable. Assume variables have a monotonic status. To compare terms headed by higher-order variables with such a status does not ensure stability by instantiation (instantiating X by, e.g., $\lambda x.c$ for some constant c), but we conjecture that it ensures a weak form of stability: $\overline{s} \succeq_{mon} \overline{t}$ implies that $\overline{s}\gamma \succeq_{mon} \overline{t}\gamma$ for all ground substitutions γ in normal form. Assuming this weaker property, we can define an improved version of $\succ_{horpo}$ by allowing such comparisons each time the non-strict ordering is used in a recursive call, that is, in cases 2a, 2d and 2c.

We believe that the obtained ordering has again all desired properties. However, the limited benefit obtained with this improvement does not seem to justify struggling with the resulting complications[4].

[3] The idea of higher-order subterm discussed by the author is very appealing per se, but we do not think that the author came up with the right technical definition.

[4] This suggestion resulted from a discussion with Jean Goubault.

8 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 consistent with the signature, a quite restrictive property indeed.

For future work, we plan to incorporate a notion of higher-order subterm to our ordering, therefore opening the way to a theory of higher-order simplification orderings. 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.

Acknowledgements: We would like to thank Daria Walukiewicz for her helpful comments on this work.

References

1. F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalization and confluence in the λ-algebraic-cube. In *Proc. 9th IEEE Symp. Logic in Computer Science, Paris*, 1993.
2. H. Barendregt. *Handbook of Theoretical Computer Science*, volume B, chapter Functional Programming and Lambda Calculus, pages 321–364. North-Holland, 1990. J. van Leeuwen ed.
3. H. Barendregt. *Handbook of Logic in Computer Science*, chapter Typed lambda calculi. Oxford Univ. Press, 1993. eds. Abramsky et al.
4. J. Van de Pol and H. Schwichtenberg. Strict functional for termination proofs. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications, Edinburgh, Great Britain*, 1995.
5. N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, March 1982.
6. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1):69–115, February 1987.
7. N. Dershowitz and J.P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–309. North-Holland, 1990.
8. R. di Cosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design*, chapter Confluence Results. Birkhäuser, Boston, 1995.
9. J.P. Jouannaud and M. Okada. Abstract data type systems. Research Report 975, Université de Paris Sud, June 1995. To appear in TCS.

10. J.P. Jouannaud and M. Okada. Inductive data type systems. Technical report, Université de Paris Sud, 1996. submitted.
11. D. Kapur and H. 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.
12. C. Loría-Sáenz. *A Theoretical Framework for Reasoning about Program Construction based on Extensions of Rewrite Systems.* PhD thesis, Fachbereich Informatik der Universität Kaiserslautern, 1993.
13. C. Loría-Sáenz and J. 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.
14. O. Lysne and J. Piris. A termination ordering for higher order rewrite systems. In *Proc. 6th Rewriting Techniques and Applications, Kaiserslautern, LNCS 914*, Kaiserslautern, Germany, 1995.
15. C. 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.
16. T. Nipkow. Higher order critical pairs. In *Proc. IEEE Symp. on Logic in Comp. Science*, Amsterdam, 1991.
17. P. Nivela and R. 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.
18. H. Zantema. Termination of term rewriting: from many-sorted to one-sorted. In Jan van Leeuwen, editor, *Computing Science in the Netherlands*, volume 2, pages 617–629, November 1991. Also appeared as report RUU-CS-91-18, Utrecht University.

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