The Higher-Order Recursive Path Ordering[*]

J.-P. Jouannaud
LRI, Bâtiment 490
UMR CNRS 8623
Université de Paris Sud
91405 Orsay, FRANCE
Jean-Pierre.Jouannaud@lri.fr

A. Rubio
Dep. LSI
Univ. Politècnica de Catalunya
Modul C6, Jordi Girona 1
08034, Barcelona, SPAIN
rubio@lsi.upc.es

Abstract

This paper extends the termination proof techniques based on reduction orderings to a higher-order setting, by adapting the recursive path ordering definition to terms of a typed lambda-calculus generated by a signature of polymorphic higher-order function symbols. The obtained ordering is well-founded, compatible with β-reductions and with polymorphic typing, monotonic with respect to the function symbols, and stable under substitution. It can therefore be used to prove the strong normalization property of higher-order calculi in which constants can be defined by higher-order rewrite rules. For example, the polymorphic version of Gödel's recursor for the natural numbers is easily oriented. And indeed, our ordering is polymorphic, in the sense that a single comparison allows to prove the termination property of all monomorphic instances of a polymorphic rewrite rule. Several other non-trivial examples are given which examplify the expressive power of the ordering.

1 Introduction

Rewrite rules are increasingly used in programming languages and logical systems, with two main goals: defining functions by pattern matching; describing rule-based decision procedures. ML, Alf [4] and Isabelle [17] examplify the first use. A future version of Coq [9] will examplify the second use [8]. In Isabelle, rules operate on terms in β-normal, η-expanded form. In ML and Alf, they operate on arbitrary terms. In the future version of Coq, both kinds should coexist.

Our ambition is to develop for the higher-order case the kind of semi-automated termination proof techniques that are available for the first-order case, of which the most popular one is the recursive path ordering [6].

Our contribution to this program is a reduction ordering for typed higher-order terms which conservatively extends Dershowitz's recursive path ordering for first-order terms. In the latter, the precedence rule allows to decrease from the term $s = f(s_1, \ldots, s_n)$ to the term $g(t_1, \ldots, t_n)$, provided that (i) f is bigger than g in the given precedence on function symbols, and (ii) s is bigger than every t_i. For typing reasons, in our ordering the latter condition becomes: (ii) for every t_i, either s is bigger than t_i or some s_j is bigger than or equal to t_i. Indeed, we can instead allow t_i to be obtained from the subterms of s by computability preserving operations. Here, computability refers to Tait and Girard's strong normalization proof technique which we have used to show that our ordering is well-founded and compatible with β-reductions.

In the litterature, one can find several attempts at designing methods for proving strong normalization of higher-order rewrite rules based on ordering comparisons. These orderings are either quite weak [13, 11], or need an important user interaction [5]. Besides, they operate on terms in η-long β-normal form, hence apply only to the higher-order rewriting "à la Nipkow" [15], based on higher-order pattern matching modulo $\beta\eta$. To our knowledge, our ordering is the first to operate on arbitrary higher-order terms, therefore applying to the other kind of rewriting, based on plain pattern matching. And indeed we want to stress four important features of our approach. First, it can be seen as a way to lift an ordinal notation operating on a first-order language (here, the set of labelled trees ordered by the recursive path ordering) to an ordinal notation of higher type operating on a set of well-typed λ-expressions built over the first-order language. Secondly, the analysis of our ordering, based on Tait and Girard's computability predicate proof technique, leads to hiding this technique away, by allowing one to carry

[*]This work was partly supported by the ESPRIT Basic Research Action CCL-II, ref. WG # 22457. and the CICYT project HEMOSS ref. TIC98-0949-C02-01

out future meta-theoretical investigations based on ordering comparisons rather than by a direct use of the computability predicate technique. Thirdly, we obtain as a biproduct a new proof of well-foundedness of Dershowitz's recursive path ordering which does not use Kruskal's tree theorem anymore. Fourthly, our ordering can be adapted to operate on terms in η-long β-normal form, yielding an ordering stronger than the already existing ones [12].

To hint at the strength of the ordering described in the present paper, let us mention that the polymorphic version of Gödel's recursor for the natural numbers is easily oriented. And indeed, our ordering can prove at once the termination property of all monomorphic instances of a polymorphic rewrite rule. Many other examples are given which examplify the expressive power of the ordering.

The framework we use is described in Section 2. The ordering is defined and studied in Section 3, where several examples are also given. The notion of computable closure used to boost the expressivity of the ordering is introduced and studied in Section 4. The discussion of the potential improvements and extensions of our work is carried out in Section 5. Some related work is mentionned in conclusion. The reader is expected to be familiar with the basics of term rewriting systems [7] and typed lambda calculi [1, 2].

2 Preliminaries

2.1 Types

Given a set $\mathcal{S}$ of *sort* constants and a set $\mathcal{S}^\forall$ of *type variables*, the set $\mathcal{T}_{\mathcal{S}^\forall}$ of *polymorphic types* is generated from these sets by the constructor $\rightarrow$ for *functional types*:

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

We will denote by $\mathcal{T}_{\mathcal{S}}$ the set of ground types. By a *basic type*, we mean a sort or a type variable. Other types are *arrow types*.

Let $\equiv$ be the congruence on types generated by equating all sorts in $\mathcal{S}$. Note that two types are equivalent iff they have the same arrow squeleton for all type instantiations.

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. A type declaration is *first-order* if it uses only sorts, and higher-order otherwise. It is *polymorphic* if it uses some non-ground type, otherwise, it is *monomorphic*. Type declarations are not types, although they are used for typing purposes.

2.2 Signatures

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 σ:

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

$\mathcal{F}$ is called a *first-order signature* if all its type declarations are first-order, and a higher-order signature otherwise. It is called a *polymorphic signature* if some type declaration is polymorphic, and a monomorphic signature otherwise. Given a *type substitution* ξ extending to types a mapping from $\mathcal{S}^\forall$ to $\mathcal{T}_{\mathcal{S}}$, we denote by $\mathcal{F}\xi$ the monomorphic signature $\{f : s_1\xi \rightarrow s_2\xi \rightarrow \ldots \rightarrow s_n\xi \mid f : s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n \in \mathcal{F}\}$. Polymorphic signatures capture infinitely many monomorphic ones via type instantiation.

2.3 Terms

The set of *untyped algebraic λ-terms* is generated from the signature $\mathcal{F}$ and 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 will usually write the application operator explicitly, as in $@(u, v)$. As a matter of convenience, we may write $u(v_1, \ldots, v_n)$, or $@(u, v_1, \ldots, v_n)$ for $u(v_1)\ldots(v_n)$, assuming $n \geq 1$. The term $@(u, \overline{v})$ is called a (partial) *left-flattening* of $s = u(v_1)\ldots(v_n)$, u being possibly an application itself.

We denote by $\mathcal{V}ar(t)$ the set of free variables of t. 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. By $|t|$ we denote de size of t, i.e. the number of symbols occurring in t, and by $|t|_v$, the size of t without counting the 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, and simply $t[\ ]_p$ for a term with a hole, also called a context. The notation $\overline{s}$ will be ambiguously used to denote a list, or a multiset, or a set of terms $s_1, \ldots, s_n$.

Substitutions are written as in $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where, for every $i \in [1..n]$, t_i is assumed different from x_i. We use the letter γ for substitutions and postfix notation for their application. 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 written $x : \sigma$, 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:
$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

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

Abstraction:
$$\frac{\Gamma \cup \{x : \sigma\} \vdash t : \tau}{\Gamma \vdash (\lambda x : \sigma . t) : \sigma \to \tau}$$

Application:
$$\frac{\Gamma \cup \{x : \sigma\} \vdash s : \sigma \to \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash s(t) : \tau}$$

Type Congruence:
$$\frac{\Gamma \vdash s : \sigma}{\Gamma \vdash s : \tau} \ \texttt{if}\ \sigma \equiv \tau$$

The last rule allows us to type terms that are usually considered as ill-typed because sort names do not match. On the other hand, strong normalisation of typed λ-calculus is clearly preserved for such terms, since it amounts to collapse all sorts into a single one. Typing these terms allows for smoother technical developments.

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 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. Note that function symbols are uncurried, hence must come along with all their arguments.

2.5 Conversion rules

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

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

As usual, we do not distinguish α-convertible terms. We use $\overset{*}{\underset{\beta}{\longleftrightarrow}}$ for the congruence generated by the β-equality, and $\longrightarrow_\beta$ for the β-reduction rule:

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

The simply typed λ-calculus is confluent and terminating (or *strongly normalizing* in the lambda-calculus jargon) with respect to β-reductions.

We could order as well η-equality as a reduction and carry it along to the price of some easy extra technical complications. We did not think it was worth doing it, since η makes more sense in the context of higher-order pattern matching modulo $\beta\eta$, a kind of higher-order rewriting that we do not consider in the present paper, as explained next.

2.6 Higher-order rewrite rules

A (possibly higher-order) *term rewriting system* is a set of rewrite rules $R = \{\Gamma_i \vdash l_i \to r_i : \sigma_i\}_i$, where l_i and r_i are higher-order 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 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 $s|_p = l\gamma$ and $t = s[r\gamma]_p$.

A term s such that $s \xrightarrow[R]{p} t$ is called *reducible* (with respect to R). $s|_p$ 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. We denote by $\xrightarrow[R]{*}$ the reflexive, transitive closure of the rewrite relation $\xrightarrow[R]{}$, and by $\longleftrightarrow^*_R$ its reflexive, symmetric, transitive closure. We are actually interested in the relation $\longrightarrow_{R\beta} = \longrightarrow_R \cup \longrightarrow_\beta$.

Given a rewrite relation $\longrightarrow$, a term s is strongly normalizing if there is no infinite sequence of rewrites issuing from s. The rewrite relation itself is *strongly normalizing*, or *terminating*, if all terms are strongly normalizing, 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.

Several examples of higher-order rewrite systems are developped in section 3.

2.7 Polymorphic Reduction Orderings

We will make intensive use of well-founded orderings for proving strong normalization properties. We will use the vocabulary of rewrite systems for orderings. For our purpose, a *strict ordering*, usually denoted by $>$, is an irreflexive and transitive relation, and an *ordering*, usually denoted by $\geq$, is the union of its strict part with α-conversion. We will essentially use strict orderings, and hence, we will use the word ordering for them too. *Rewrite orderings* are *monotonic* and *stable* orderings, and *reduction orderings* are in addition *well-founded*. Monotonicity of $>$ is defined as $u > v$ implies $s[u]_p > s[v]_p$ for all contexts $s[\ ]_p$. Stability of $>$ is defined as $u > v$ implies $s\gamma > t\gamma$ for all substitutions γ. 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 $>_1, \ldots, >_n$ are well-founded orderings on sets $S_1, \ldots, S_n$. Then $(>_1, \ldots, >_n)_{lex}$ is a well-founded ordering on $S_1 \times \ldots \times S_n$. We write $>_{lex}$ if all sets and orderings are equal.

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):

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

We end up with the definition of a polymorphic reduction ordering operating on higher-order terms, and allowing one's to show that the relation $\longrightarrow_R \cup \longrightarrow_\beta$ is well-founded by simply comparing the lefthand and righthand sides of the (polymorphic) rules in R:

Definition 2.1 *An ordering $>$ is* polymorphic *if $s > t$ in a polymorphic signature $\mathcal{F}$ implies $s > t$ in all monomorphic instances $\mathcal{F}\xi$ of $\mathcal{F}$.*

A polymorphic higher-order reduction ordering is a monotonic and stable polymorphic ordering $>$ of the set of higher-order terms, such that $> \cup \longrightarrow_\beta$ is well-founded.

3 The Higher-Order Recursive Path Ordering

From now on, the application operator is written explicitly. Let $Mul \uplus Lex$ be a partition of $\mathcal{F}$, s.t. all symbols in Lex have a fixed arity, and $>_\mathcal{F}$ be a well-founded ordering on $\mathcal{F}$, called the *precedence*. As usual, variables will be considered as constants incomparable among themselves and with other function symbols.

3.1 Definition of the ordering

The following higher-order recursive path ordering (HORPO) is essentially the same as Dershowitz's recursive path ordering for first-order terms. The only difference is that we take care of higher-order arguments on the smaller side by having a corresponding bigger higher-order argument on the bigger side.

Definition 3.1 $s : \sigma \succ_{horpo} t : \tau$ *iff* $\sigma \equiv \tau$ *and*

1. $s = f(\overline{s})$ *with* $f \in \mathcal{F}$, *and* $s_i \succeq_{horpo} t$ *for some* $s_i \in \overline{s}$.
2. $f, g \in \mathcal{F}$, $f >_\mathcal{F} g$ *and* $t = g(\overline{t})$ *and* $\forall t_i \in \overline{t}$ $s \succ_{horpo} t_i$ *or* $s_j \succeq_{horpo} t_i$ *for some* $s_j \in \overline{s}$
3. $f = g \in Mul$ *and* $\overline{s}(\succ_{horpo})_{mul}\overline{t}$
4. $f = g \in Lex$ *and* $\overline{s}(\succ_{horpo})_{lex}\overline{t}$, *and* $\forall t_i \in \overline{t}$ $s \succ_{horpo} t_i$ *or* $s_j \succeq_{horpo} t_i$ *for some* $s_j \in \overline{s}$
5. $f \in \mathcal{F}$, $@(\overline{t})$ *is some partial left-flattening of* t, *and* $\forall t_i \in \overline{t}$ $s \succ_{horpo} t_i$ *or* $s_j \succeq_{horpo} t_i$ *for some* $s_j \in \overline{s}$
6. $s = @(s_1, s_2)$, $t = @(t_1, t_2)$ *and* $\{s_1, s_2\}(\succ_{horpo})_{mul}\{t_1, t_2\}$
7. $s = \lambda x.u$, $t = \lambda x.v$ *and* $u \succ_{horpo} v$

Important observations are the following:

Our ordering can only compare terms with equivalent types.

When the signature is first-order, cases 1, 2, 3, and 4 reduce to the usual recursive path ordering for first-order terms.

Case 7 compares two abstractions. No other case applies to an abstraction on the bigger side. Case 1 may compare a term headed by a function symbol on the bigger side with an abstraction on the smaller side.

In Case 6, type considerations show that the first arguments are compared together as well as the second's. On the other hand, Case 5 compares a term s headed by a function symbol with an application on the smaller side, for which an appropriate (partial) left-flattening has to be chosen non-deterministically. This non-deterministic choice is essential for stability. Type considerations may help finding one quickly.

In cases 2, 4, and 5, the non-deterministic or comparison can be replaced by the equivalent deterministic one:
if $t_i : \rho \equiv \tau$ then $s \succ_{horpo} t_i$
otherwise $s_j \succeq_{horpo} t_i$ for some $s_j : \rho \in \overline{s}$ such that $\rho \equiv \sigma$

Example 1 Let $\mathcal{S} = \{\mathbb{N}\}$, $\mathcal{S}^\forall = \{\alpha\}$ and $\mathcal{F} = \{0 : \mathbb{N},\ s : \mathbb{N} \to \mathbb{N},\ rec : \mathbb{N} \times \alpha \times (\mathbb{N} \to \alpha \to \alpha) \to \alpha\}$. Gödel's recursor for natural numbers is defined by the following rewrite rules:

$$\begin{array}{rcl} rec(0, u, X) & \to & u \\ rec(s(x), u, X) & \to & @(X, x, rec(x, u, X)) \end{array}$$

The first rule succeeds immediately by case 1. For the second rule, we apply case 5, and need to show recursively that (i) $X \succeq_{horpo} X$, (ii) $s(x) \succ_{horpo} x$, and (iii) $rec(s(x), u, X) \succ_{horpo} rec(x, u, X)$. (i) is trivial. (ii) is by case 1. (iii) is by case 3, calling again recursively for $s(x) \succ_{horpo} x$.

Note that we have proved Gödel's polymorphic recursor, for which the output type of rec is any given type. This is because we do not care about types in our comparisons, provided two compared terms are typable with equivalent types, hence of the same functionnal structure.

We can of course now add some defining rules for sum and product:

$$\begin{array}{rcl} x+0 & \rightarrow & 0 \\ x+s(y) & \rightarrow & s(x+y) \\ x*y & \rightarrow & rec(y,0,\lambda z_1 z_2.x+z_2) \end{array}$$

The first two first-order rules are easily taken care of. For the third, we use the precedence $* >_{\mathcal{F}} rec$ to eliminate the rec operator. But the computation fails, since there is no higher-order subterm of $x*y$ to take care of the righthand side subterm $\lambda z.x+z$. We will come back to this example in Section 4.

Example 2 Let $\mathcal{S} = \{List, \mathbb{N}\}$ and $\mathcal{F} = \{cons : \mathbb{N} \times List \rightarrow List,\ map : List \times (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow List\}$. The rules for map are:

$$\begin{array}{rcl} map(nil, X) & \rightarrow & nil \\ map(cons(x,l), X) & \rightarrow & cons(@(X,x), map(l,X)) \end{array}$$

The first rule is trivially taken care of by case 1. For the second, let $map \in Mul$ and $map >_{\mathcal{F}} cons$. Since $map >_{\mathcal{F}} cons$, applying case 2, we need to show that $map(cons(x,l),X) \succ_{horpo} @(X,x)$ and $map(cons(x,l),X) \succ_{horpo} map(l,X)$. The latter is true by case 3, since $cons(x,l) \succ_{horpo} l$ by case 1. The first is by case 5 (note the use of the equivalence on sorts), as X is an argument of the first term and $cons(x,l) \succ_{horpo} x$ by case 1.

Considering now a polymorphic version of the same example, with $\mathcal{S} = \{List\}$, $\mathcal{S}^{\forall} = \{\alpha\}$, and $\mathcal{F} = \{cons : \alpha \rightarrow List \rightarrow List,\ map : List \times (\alpha \rightarrow \alpha) \rightarrow List\}$, then the computation fails since $X(x)$ has now type α which cannot be compared to the sort $List$. Note, however, that we could deal with all type instantiations replacing α by an arbitrary sort. We will discuss this example further in Section 5. 4.

3.2 Properties of the higher-order recursive path ordering

Theorem 3.2 $\succ_{horpo}$ *is a decidable, polymorphic, higher-order reduction ordering.*

Monotonicity, stability under substitutions and polymorphism are proved by induction on the size of terms. Although the proofs are slightly more difficult technically than the usual proofs for the recusive path ordering, they follow the same kind of pattern (polymorphism was actually never considered before). This contrasts with the proof of well-foundedness and compatibility with β-reductions, for which we need to show that the relation $\longrightarrow\ = \succ_{horpo} \cup \longrightarrow_{\beta}$ is strongly normalizing.

The well-foundedness of the recursive path ordering is usually proved by using the fact that it contains the embedding relation which is a well-order by Kruskal's tree theorem. Since we do not know of any non-trivial extension of Kruskal's tree theorem for higher-order terms that includes β-reductions, we will adopt a completely different method, the computability predicate proof method due to Tait and Girard. As a bi-product, we obtain a new proof of well-foundedness for the recursive path ordering on first-order terms. Indeed, this proof method will also suggest improvements of our ordering, that will be discussed in Section 4. We give no more than the necessary details for the understanding of the reader.

Our definition of computability for typed terms is standard:

Definition 3.3 *An algebraic λ-term s is* computable *iff*

(i) s is a variable, or

(ii) s has a basic type and s is strongly normalizable, or else

(iii) s has an arrow type $\sigma \rightarrow \tau$ and $@(s,t)$ is computable for every computable term t of type σ.

A term is neutral *if it is not an abstraction. A vector $\overline{s}$ of terms is computable iff so are all its components.*

We first recall the properties of the computability predicate.

Property 3.4 (Computability Properties)

(i) Every computable term is strongly normalizable.

(ii) Assume that t is computable and $t \longrightarrow s$. Then s is computable.

(iii) A neutral term t is computable iff s is computable for all s such that $t \longrightarrow s$.

(iv) Let $\overline{t}$ be a vector of at least two computable terms. Then $@(\overline{t})$ is computable.

(v) $\lambda x : \sigma.u$ is computable iff $u\{x \mapsto w\}$ is computable for every computable term $w : \sigma$.

(vi) Let t be a term of sort type. Then, t is computable iff it is strongly normalizable.

Proof: All proofs are standard, but the one of (v). The only if part is Property 3.4 (ii). For the converse, we prove that $@(\lambda x.u, w)$ is computable for an arbitrary term $w : \sigma$ if $u\{x \mapsto w\}$ and w are computable.

Since variables are computable, $u = u\{x \mapsto x\}$ is computable by assumption. By property (i), u and w are strongly normalizable. We prove that $@(\lambda x.u, w)$ is computable by induction on the set $\{u, w\}$ ordered by $(\longrightarrow)_{mul}$.

By (iii), the neutral term $@(\lambda x.u, w)$ is computable iff v is computable for all v such that $@(\lambda x.u, w) \longrightarrow v$. There are several cases to be considered.

1. Let $v = @(\lambda x.u', w)$ with $u \longrightarrow u'$. Then, $u\{x \mapsto w\} \longrightarrow u'\{x \mapsto w\}$. Hence, by assumption and (ii), $u'\{x \mapsto w\}$ is computable for every computable $w : \sigma$. Since $\{u, w\}(\longrightarrow)_{mul}\{u', w\}$, by induction hypothesis, v is computable.

2. Let $v = @(\lambda x.u, w')$ with $w \longrightarrow_\beta w'$. Then $\{u, w\}(\longrightarrow)_{mul}\{u, w'\}$, and by induction hypothesis, v is computable.

3. $v = u\{x \mapsto w\}$ is computable by assumption.

4. Let $t = @(\lambda x.u, w) \succ_{horpo} v = @(v', w')$ by case 6. For typing reasons, $w \succeq_{horpo} w'$ and $\lambda x.u \succeq_{horpo} v'$. Hence, w' is computable by (ii). On the other hand, by property of the ordering, $v' = \lambda x.u'$ with $u \succeq_{horpo} u'$. Now, since $u\{x \mapsto w\}$ is computable for every computable $w : \sigma$ by assumption, $u'\{x \mapsto w\}$ is computable for every computable $w : \sigma$ by (ii). Hence $\lambda x.u'$ is computable, and therefore v is computable. $\square$

The following lemma and proof are both essential:

Property 3.5 *Let $f \in \mathcal{F}$ and let $\bar{t}$ be a set of terms. If $\bar{t}$ is computable, then $f(\bar{t})$ is computable.*

Proof: Let $t_i : \tau_i$ and $f(\bar{t}) : \tau$. Since terms in $\bar{t}$ are computable, by Property 3.4 (i), they are strongly normalizable. We use this remark to build our induction argument: we prove that $f(\bar{t})$ is computable by induction on the pair $\langle f, \bar{t}\rangle$ ordered lexicographically by $(>_{\mathcal{F}}, (\longrightarrow)_{stat})_{lex}$ where $stat$ is either mul or lex, depending on the symbol f.

Since $f(\bar{t})$ is neutral, by Property 3.4 (iii), it is computable iff every s such that $f(\bar{t}) \longrightarrow s$ is computable, which we prove by an inner induction on the size of s.

Let us assume first that $s = f(t_1, \ldots, t_i', \ldots, t_n)$, with $t_i \longrightarrow_\beta t_i'$. By Property 3.4 (ii), t_i' is computable, and since $\bar{t}(\longrightarrow)_{stat}\bar{s}$ for any status $stat$, s is computable by application of the outer induction hypothesis. We are left with the cases corresponding to the application of $\succ_{horpo}$ to the term s.

1. Let $f(\bar{t}) \succ_{horpo} s$ by case 1, hence $t_i \succeq_{horpo} s$ for some $t_i \in \bar{t}$. Since t_i is computable, s is computable by Property 3.4 (ii).

2. Let $t = f(\bar{t}) \succ_{horpo} s$ by case 2. Then $s = g(\bar{s})$, $f >_{\mathcal{F}} g$ and for every $s_i \in \bar{s}$ either $t \succ_{horpo} s_i$, in which case s_i is computable by the inner induction hypothesis, or $t_j \succeq_{horpo} s_i$ for some $t_j \in \bar{t}$ and s_i is computable by Property 3.4 (ii). Therefore, $\bar{s}$ is computable, and since $f >_{\mathcal{F}} g$, s is computable by the outer induction hypothesis.

For the following two cases, $s = f(\bar{s})$, with $s_i : \tau_i$.

3. If $f(\bar{t}) \succ_{horpo} s$ by case 3, then $\bar{t}(\succ_{horpo})_{mul}\bar{s}$. By definition of the multiset comparison, for every $s_i \in \bar{s}$ there is some $t_j \in \bar{t}$, s.t. $t_j \succeq_{horpo} s_i$, hence, by Property 3.4 (ii), s_i is computable. This allows us to conclude by the outer induction hypothesis that s is computable.

4. If $f(\bar{t}) \succ_{horpo} s$ by case 4, then $\bar{t}(\succ_{horpo})_{lex}\bar{s}$ and for every $s_i \in \bar{s}$ or $t_j \succeq_{horpo} s_i$ for some $t_j \in \bar{t}$. As in the precedence case, this implies that $\bar{s}$ is computable. Then, since $\bar{t}(\succ_{horpo})_{lex}\bar{s}$, s is computable by the outer induction hypothesis.

5. If $f(\bar{t}) \succ_{horpo} s$ by case 5, let $@(s_1, \ldots, s_n)$ be the partial left-flattening of s used in that proof. By the same token as in case 2, every term in $\bar{s}$ is computable, hence s is computable by Property 3.4 (iv). $\square$

Lemma 3.6 *Let γ be a computable substitution and t be an algebraic λ-term. Then $t\gamma$ is computable.*

Proof: The proof proceeds by induction on $|t|_v$.

1. t is a variable x. Then $x\gamma$ is computable by assumption.

2. t is an abstraction $\lambda x.u$. By Property 3.4 (v), $t\gamma$ is computable if $u\gamma\{x \mapsto w\}$ is computable for every well-typed computable term w. Taking $\delta = \gamma \cup \{x \mapsto w\}$, we have $u\gamma\{x \mapsto w\} = u(\gamma \cup \{x \mapsto w\})$ since x may not occur in γ. Since δ is computable, and $|t|_v > |u|_v$, by induction hypothesis, $u\delta$ is computable.

3. $t = @(t_1, t_2)$ or $t = f(t_1, \ldots, t_n)$, and some t_i is not a variable. Let $s = f(x_1, \ldots, x_n)$ or $s = @(x_1, x_2)$ and $\delta = \{x_i \mapsto t_i \mid 1 \leq i \leq n\}$. By induction hypothesis $t_i\gamma$ is computable for all i, hence δ is computable. Since not all t_i are variables, $|t|_v > |s|_v$ and, by induction hypothesis $s\delta$ is computable.

4. $t = @(x_1, x_2)$. Then $t\gamma$ is computable by Property 3.4 (iv).

5. $t = f(x_1, \ldots, x_n)$. Then $t\gamma$ is computable by Property 3.5. $\square$

We can now easily conclude the proof of well-foundedness and compatibility with β-reductions needed for our main theorem, by showing that every term is strongly normalizable with respect to $\longrightarrow$. Given an arbitrary term t, let γ be the identity substitution. Since γ is computable, $t = t\gamma$ is computable by Lemma 3.6, and strongly normalizable by Property 3.4 (i).

The following example gives a set of rewrite rules defining the insertion algorithm for the (ascending or descending) sort of a list of natural numbers.

Example 3 Insertion Sort. Let $\mathcal{S} = \{\mathbb{N}, List\}$ and $\mathcal{F} = \{nil : List; cons : \mathbb{N} \times List \to List; max, min : \mathbb{N} \times \mathbb{N} \to \mathbb{N}; insert : \mathbb{N} \times List \times (\mathbb{N} \times \mathbb{N} \to \mathbb{N}) \times (\mathbb{N} \times \mathbb{N} \to \mathbb{N}) \to List; sort : List \times (\mathbb{N} \times \mathbb{N} \to \mathbb{N}) \times (\mathbb{N} \times \mathbb{N} \to \mathbb{N}) \to List; ascending\_sort, descending\_sort : List \to List\}$.

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

We simply need the precedence $max, min >_{\mathcal{F}} 0$ for these first-order rules.

$$\begin{array}{rcl} insert(n, nil, X, Y) & \rightarrow & cons(n, nil) \\ insert(n, cons(m,l), X, Y) & \rightarrow & \\ \multicolumn{3}{r}{cons(X(n,m), insert(Y(n,m), l, X, Y))} \end{array}$$

The first $insert$ rule is easily taken care of by applying case 2 with the precedence $insert >_{\mathcal{F}} cons$, and then case 1. For the second $insert$ rule, we apply first case 2, and we recursively need to show: firstly, that $insert(n, cons(m,l), X, Y) \succ_{horpo} @(X, n, m)$, which follows by applying rule 2, and then case 1 recursively; and secondly that $insert(n, cons(m,l), X, Y) \succ_{horpo} insert(Y(n,m), l, X, Y)$, which succeeds as well by case 4, with a right-to-left lexicographic status for $insert$, and calling recursively with $insert(n, cons(m,l), X, Y) \succ_{horpo} @(Y, n, m)$, which is solved by case 6.

$$\begin{array}{rcl} sort(nil, X, Y) & \rightarrow & nil \\ sort(cons(n,l), X, Y) & \rightarrow & \\ \multicolumn{3}{r}{insert(n, sort(l, X, Y), X, Y)} \end{array}$$

Again, these rules are easily oriented by $\succ_{horpo}$, by using the precedence $sort >_{\mathcal{F}} insert$. On the other hand, $\succ_{horpo}$ fails to orient the following two seemingly easy rules.

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

This is so, because the term $\lambda xy.min(x,y)$ occuring in the lefthand side has type $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$, which is not comparable to any lefthand side type. We will come back to this example in Section 4.

4 Computational Closure

The ordering is quite sensitive to innocent variations of the language, like adding (higher-order) dummy arguments to righthand sides, or η-converting expressions. We will now solve these problems by improving our definition in the light of the strong normalization proof. In that proof, it was crucial to show the computability of the righthand side subterms by using the lefthand side subterms. In our definition, we actually require that for each righthand side subterm v, there exists a lefthand side subterm u such that $u \succeq_{horpo} v$. Assuming u is computable, then v is computable by Property 3.4 (ii). But any computability preserving operation applied to the lefthand side subterms in order to construct a term of the appropriate type would do as well. For example, the higher-order variable X is computable if and only if $\lambda x.X(x)$ is computable. Therefore, both forms may coexist. This discussion is formalized below with the notion of a computational closure adapted from [3].

Definition 4.1 *Given a term $t = f(\overline{t})$, we define its* computable closure *$\mathcal{CC}(t)$ as $\mathcal{CC}(t, \emptyset)$, where $\mathcal{CC}(t, \mathcal{V})$, with $\mathcal{V} \cap \mathcal{V}ar(t) = \emptyset$, is the smallest set of well-typed terms containing all variables in $\mathcal{V}$ and all terms in $\overline{t}$, as well as their subterms $u : \tau$ such that τ is a sort and $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$, and closed under the following operations:*

1. *precedence: let g such that $f >_{\mathcal{F}} g$, and $\overline{s} \in \mathcal{CC}(t, \mathcal{V})$; then $g(\overline{s}) \in \mathcal{CC}(t, \mathcal{V})$;*
2. *recursive call: let $\overline{s}$ be a sequence of terms in $\mathcal{CC}(t, \mathcal{V})$ such that $f(\overline{s})$ is well typed, $\mathcal{V}ar(\overline{s}) \subseteq \mathcal{V}ar(t)$ and $\overline{t}(\longrightarrow_{\beta} \cup \rhd)_{stat_f} \overline{s}$; then $f(\overline{s}) \in \mathcal{CC}(t, \mathcal{V})$;*
3. *application: let $s : \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma \in \mathcal{CC}(t, \mathcal{V})$ and $u_i : \sigma_i \in \mathcal{CC}(t, \mathcal{V})$ for every $i \in [1..n]$; then $@(s, u_1, \ldots, u_n) \in \mathcal{CC}(t, \mathcal{V})$;*
4. *abstraction: let $x \notin \mathcal{V}ar(t) \cup \mathcal{V}$ and $s \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$; then $\lambda x.s \in \mathcal{CC}(t, \mathcal{V})$;*
5. *reduction: let $u \in \mathcal{CC}(t, \mathcal{V})$, and $u \longrightarrow_{\beta} v$; then $v \in \mathcal{CC}(t, \mathcal{V})$.*

Note the conditions $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$ in the basic case, and $\mathcal{V}ar(\overline{s}) \subseteq \mathcal{V}ar(t)$ in case 2. These two conditions are one of the main weaknesses of our definition.

Apart from the reduction rule, membership to the computation closure of a term may not be decidable. On the other hand, it becomes easily decidable if the use of the reduction rule is bounded.

The following property can easily be shown by induction on the definition of the computable closure:

Lemma 4.2 *Assume that $u \in \mathcal{CC}(t)$. Then, $u\gamma \in \mathcal{CC}(t\gamma)$ for every substitution γ.*

We can now modify our ordering as follows:

Definition 4.3 *$s : \sigma \succ_{horpo} t : \tau$ iff $\sigma \equiv \tau$ and*

1. *$s = f(\overline{s})$ with $f \in \mathcal{F}$, and $s_i \succeq_{horpo} t$ for some $s_i \in \overline{s}$.*
2. *$f, g \in \mathcal{F}$, $f >_{\mathcal{F}} g$ and $t = g(\overline{t})$ and*
   $\forall t_i \in \overline{t}$ either $s \succ_{horpo} t_i$ or $s_j \succeq_{horpo} t_i$ for some $s_j \in \overline{s}$ or $t_i \in \mathcal{CC}(s)$
3. *$f = g \in Mul$ and $\overline{s}(\succ_{horpo})_{mul}\overline{t}$*

4. *$f = g \in Lex$ and $\overline{s}(\succ_{horpo})_{lex}\overline{t}$, and $\forall t_i \in \overline{t}$ either $s \succ_{horpo} t_i$ or $s_j \succeq_{horpo} t_i$ for some $s_j \in \overline{s}$ or $t_i \in \mathcal{CC}(s)$*

5. *$f \in \mathcal{F}$, $@(\overline{t})$ is some partial left-flattening of t, and $\forall t_i \in \overline{t}$ either $s \succ_{horpo} t_i$ or $s_j \succeq_{horpo} t_i$ for some $s_j \in \overline{s}$ or $t_i \in \mathcal{CC}(s)$*

6. *$s = @(s_1, s_2)$, $t = @(t_1, t_2)$ and $\{s_1, s_2\}(\succ_{horpo})_{mul}\{t_1, t_2\}$*

7. *$s = \lambda x.u$ and $t = \lambda x.v$ and $u \succ_{horpo} v$*

As previously, the or occuring in the clause $s \succ_{horpo} t_i$ or $s_j \succeq_{horpo} t_i$ for some $s_j \in \overline{s}$ or $t_i \in \mathcal{CC}(s)$ is non-deterministic. Again, we can make it deterministic by approriately ordering the clauses.

The following lemma is easy to prove:

Lemma 4.4 *$\succ_{horpo}$ is monotonic, stable, and polymorphic.*

We now show that terms in the computable closure of a term are computable under the appropriate hypotheses for its use. For this, we first remark that the computability properties are still valid, without any change in the proofs.

Property 4.5 *Assume $\overline{t} : \overline{\tau}$ is computable, as well as every term $g(\overline{s})$ with $\overline{s}$ computable and $g(\overline{s})$ smaller than $t = f(\overline{t})$ in the ordering $\langle >_{\mathcal{F}}, (\longrightarrow \cup \rhd)_{stat}\rangle$ operating on pairs $\langle f, \overline{t}\rangle$ where stat is the status of f. Then, every term in $\mathcal{CC}(t)$ is computable.*

The precise formulation of this statement arises from its forthcoming use inside the proof of Lemma 4.6. Note that we have increased the ordering $(\longrightarrow)_{stat}$ used in the proof of lemma 3.5 by adding strict subterm, yielding the ordering $(\longrightarrow \cup \rhd)_{stat}$. This is possible, because the latter is well-founded on a given set of terms when so is the former, since $\longrightarrow$ has just be shown to be monotonic, see [7] for the argument.

Proof: We prove that $u\gamma$ is computable for every computable substitution γ of domain $\mathcal{V}$ and every $u \in \mathcal{CC}(t, \mathcal{V})$ such that $\mathcal{V} \cap \mathcal{V}ar(t) = \emptyset$. We obtain the result by taking $\mathcal{V} = \emptyset$. The proof is by induction on the definition of the computational closure. For the basic case: if $u \in \overline{t}$, we conclude by assumption that $\overline{t}$ is computable since $u\gamma = u$ by assumption on $\mathcal{V}$; if $u \in \mathcal{V}$, then $u\gamma$ is computable by assumption on γ; otherwise, u is a subterm of type a sort of some $v \in \overline{t}$. As before, $v\gamma = v$ is computable, hence is strongly normalizable by Property 3.4 (i). By definition of the closure, $u\gamma = u$, therefore, since $v\gamma$ is strongly normalizable, by monotonicity, $u\gamma$ is also strongly normalizable, and hence computable by Property 3.4 (vi). We now discuss the successive operations to form the closure.

1. case 1: $u = g(\overline{u})$ where $\overline{u} \in \mathcal{CC}(t, \mathcal{V})$. By induction hypothesis, $\overline{u}\gamma$ is computable, and, by assumption, since $f >_{\mathcal{F}} g$, $u\gamma$ is computable.

2. case 2: $u = f(\overline{u})$ where $\overline{u} \in \mathcal{CC}(t, \mathcal{V})$ and $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$. By induction hypothesis, $\overline{u}\gamma = \overline{u}$ is computable, and, by assumption, since $\overline{t}(\longrightarrow_{\beta} \cup \rhd)_{stat_f}\overline{u}$, $u = u\gamma$ is computable.

3. case 3: by induction and Property 3.4 (iv).

4. case 4: let $u = \lambda x.s$ with $x \notin \mathcal{V}ar(t) \cup \mathcal{V}$ and $u \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$. By induction hypothesis, $s(\gamma \cup \{x \mapsto w\})$ is computable for an arbitrary computable w, and by Property 3.4 (v), $(\lambda x.s)\gamma$ is computable.

5. case 5. By Property 3.4 (ii). □

We now restate Property 3.5 and show in one case how the proof makes use of Property 4.5.

Property 4.6 *Let $f \in \mathcal{F}$ and let $\overline{t}$ be a set of terms. If $\overline{t}$ is computable, then $f(\overline{t})$ is computable.*

Proof: We prove that $f(\overline{t})$ is computable by an outer induction on the pair $\langle f, \overline{t}\rangle$ ordered lexicographically by the ordering $(>_{\mathcal{F}}, (\longrightarrow \cup \rhd)_{stat})_{lex}$ introduced in Property 4.5, and an inner induction on the size of the reducts of t. Since the proof is similar to the proof of Property 3.5, we do only case 2.

2. Let $t = f(\overline{t}) \succ_{horpo} s$ by case 2. Then $s = g(\overline{s})$, $f >_{\mathcal{F}} g$ and for every $s_i \in \overline{s}$ either $t \succ_{horpo} s_i$ or $t_j \succeq_{horpo} s_i$ for some $t_j \in \overline{t}$ and s_i is computable as in the proof of Property 3.5, or $s_i \in \mathcal{CC}(t)$, and then, by induction hypothesis, we can apply Property 4.5 and conclude that s_i is computable. □

Theorem 4.7 *$\succ_{horpo}$ is included in a polymorphic higher-order reduction ordering.*

Proof: Thanks to Property 4.5, the strong normalization proof of this improved ordering is exactly the same as previously. Then by Lemma 4.4 we conclude that the transitive closure of $\succ_{horpo}$ is a higher-order polymorphic reduction ordering. □

This new definition is much stronger than the previous one. In addition to prove the strong normalization property of the remaining rules of the sorting example, and for the same reason, we can easily also add the following rule to the other rules of Example 1:

Example 4 $\quad n * m \quad \rightarrow \quad rec(n, 0, \lambda z_1 z_2.m + z_2)$

This additional rule can be proved terminating with the precedence: $* >_{\mathcal{F}} \{rec, +, 0\}$, since $\lambda z_1 z_2.m + z_2 \in CS(n * m)$: by base case, m and z_2 belong to $\mathcal{CC}(n * m, \{z_1, z_2\})$, hence $m + z_2 \in \mathcal{CC}(n * m, \{z_1, z_2\})$ by case 1 of the definition of the computational closure. Applying case 4 twice yields then the result.

We end this section with two examples showing the current limits of our ordering. The first one is an example about process algebra, in which a quantifier (Σ) binds variables via the use of a functional argument, that is, an abstraction:

Example 5 (taken from [5]) Let $\mathcal{S} = \{proc, data\}$, $\mathcal{F} = \{+ : proc \times proc \to proc, \cdot : proc \times proc \to proc, \delta : proc, \Sigma : (proc \to proc) \to proc)\}$. The rules are the following:

$$\begin{array}{rcl}
x + x & \to & x \\
(x + y) \cdot z & \to & (x \cdot z) + (y \cdot z) \\
(x \cdot y) \cdot z & \to & x \cdot (y \cdot z) \\
x + \delta & \to & x \\
\delta \cdot x & \to & \delta \\
\Sigma(\lambda y.x) & \to & x \\
\Sigma(\lambda y.P(y)) + P(D) & \to & \Sigma(\lambda y.P(y)) \\
\Sigma(\lambda y.P(y) + Q(y)) & \to & \Sigma(\lambda y.P(y)) + \Sigma(\lambda y.Q(y)) \\
\Sigma(\lambda y.P(y)) \cdot x & \to & \Sigma(\lambda y.P(y) \cdot x)
\end{array}$$

All rules but the last one can be oriented. This is quite surprising, since due to the fact that no application occurs in the example, it seems that proving its termination should be simple. We will come back to this example in Section 5. 3.

Example 6 (inspired by [13]) Let $\mathcal{S} = \{List\}, \mathcal{S}^{\forall} = \{\alpha\}$, $\mathcal{F} = \{nil : List, cons : \alpha \times List \to List, fcons : (\alpha \to \alpha) \to List, dapply : \alpha \times (\alpha \to \alpha) \times (\alpha \to \alpha) \to \alpha, lapply : \alpha \times List \to \alpha\}$. The rules are the following:

$$\begin{array}{rcl}
dapply(x, F, G) & \to & F(G(x)) \\
lapply(x, nil) & \to & x \\
lapply(x, cons(F, L)) & \to & F(lapply(x, L))
\end{array}$$

We can easily prove the first two rules, which do not even need the closure. For the third, we face the problem that F is not in the closure of the lefthand side, as a higher-order variable occuring inside a strict subterm of a lefthand side argument. This is important, as shown in [10], where a non-terminating example is constructed by using such a deep variable. We will come back to this problem in Section 5. 4.

5 Discussion

Our definition of the higher-order recursive path ordering can be improved again in several different ways that we discuss in turn.

1. By replacing the condition $s_j \succeq_{horpo} t_i$ for some $s_j \in \overline{s}$ or $t_i \in \mathcal{CC}(s)$ in cases 2 and 4 of the definition of the improved higher-order recursive path ordering by the weaker condition $u \succeq_{horpo} t_i$ for some $u \in \mathcal{CC}(s)$. This extension does not complicate the proofs, but seems of little practical value while increasing the complexity of the computation.

2. By replacing the condition $u \longrightarrow_{\beta} v$ in case 5 of the definition of the computable closure, by the weaker condition $u \longrightarrow_{\beta} \cup \succ_{horpo} v$. This variant will result in important complications in the proofs, since, with this modification, the ordering and the computational closure become mutually inductive definitions. We plan to study it for two reasons. Firstly, because it *may* yield a transitive relation; transitivity is essentially a matter of aesthetics, of course, since well-foundedness does not rely on transitivity. Secondly, and this second reason is more important, because of the potential practical improvement that it may provide.

3. In our ordering, all arrow type terms are treated as if they could become applied and serve in a β-reduction. This makes it difficult to prove the termination of rules whose righthand side has arrow type subterms which do not occur as lefthand side arguments. In such a case, the use of the computable closure may sometimes help, but Example 5 shows that this is not always the case. In this example, the righthand side arrow type subterm $\lambda y.P(y) \cdot x$ is treated as if it could be applied, but it will never be. We have already studied a solution to this problem that will be part of a full version of this paper. The idea is to split the set of function symbols in two subsets: those for which case 6 can be used (that is, which are greater than application) and all the others. For the first subset, the same condition as now for its arrow type subterms must be required, since by applying case 6 this subterms can occur as first argument of an application; for the second subset, however, this will never occur and hence the conditions can be relaxed.

4. The set of types is generated from a set of sort constants. We plan to use an algebra of sort terms instead, allowing us to have sort constructors, resulting in polymorphic data structures such as lists of elements of an arbitrary type. This should allow us to extend the computable closure of a term t whose type is a sort term s, by taking any immediate subterm u whose type is a subterm of s. For an example, the higher-order variable F of example 6 would be in the closure of $lapply(x, cons(F, L))$. We even believe that any subterm of t whose type is a subterm of s would do, but this has to be checked.

5. As important, is the development of an ordering able to compare terms of the calculus of constructions. Such an ordering is easily obtained by adding a new clause ensuring monotonicity of dependent products, in the same way as case 7 for abstractions. The difficulty then is to show the compatibility of the ordering with the reduction relation of the calculus, because the ob-

tained extension of the calculus of constructions becomes dependent on the ordering via its conversion rule. As a consequence, it becomes impossible to separate strong normalization, confluence and subject reduction. Since similar problems have been encountered in the past when investigating other extensions of the calculus of constructions, such an extension should not be out of hands.

6. Another important question is whether these techniques apply for proving termination of rules operating on higher-order terms in η-long β-normal form. We have already studied the question, and the answer is affirmative. As an aplication of the present work, we have obtained such an ordering which significantly improves the few attempts that we know of [13, 14, 11]. In particular, none of them was able to order automatically the recursor for natural numbers, which we can do easily. On the other hand, the obtained ordering is not fully polymorphic as the one described here, and we are still working on an improved fully polymorphic version.

7. Finally, we plan to apply the same technique to other orderings operating on first-order terms, either to variants of the recursive path ordering such as its associative commutative versions [19], or to stronger ordinal notations, such as quasi ordinal diagrams [16]. We believe that the first extension should not be too difficult. The second is likely to work as well, since we do not rely on the subterm property for the strong normalization proof.

6 Conclusion

We have defined a powerful mecanism for defining orderings operating on higher-order terms. Based on the notion of the computable closure of a term, we have succeeded to define a conservative extension of Dershowitz's recursive path ordering, which is indeed a polymorphic reduction ordering compatible with β-reductions. To our knowledge, this is the first such ordering ever.

The idea of the computable closure is also used in a different context in [3]. The goal there is to define a syntactic class of higher-order rewrite rules that are compatible with beta reductions and with recursors for arbitrary positive inductive types. The language is indeed the calculus of inductive constructions generated by a monomorphic signature. The usefulness of the notion of closure in this different context shows the strength of this concept.

Acknowledgments: We thank the referees and our close colleagues whose questions prompted us to making our main definition more understandable.

References

[1] 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.

[2] H. Barendregt. *Handbook of Logic in Computer Science*, chapter Typed lambda calculi. Oxford Univ. Press, 1993. Eds. Abramsky et al.

[3] F. Blanqui, J.-P. Jouannaud, and M. Okada. The Calculus of Algebraic Constructions. In *Tenth Int. Conf. on Rewriting Techniques and Applications*. Trento, 1999. To appear in LNCS, Springer-Verlag.

[4] T. Coquand, B. Nordstr¨om, J. M. Smith and B. von Sydow. Type Theory and Programming. In *EATCS Bulletin*, pages 203–228, 1994.

[5] J. Van de Pol and H. Schwichtenberg. Strict functionals for termination proofs. In *Typed Lambda Calculi and Applications*. Edinburgh, 1995. LNCS 902, Springer-Verlag.

[6] N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[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] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo, 1998. Rapport de recherche INRIA 3400.

[9] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin-Mohring, and B. Werner. The coq proof assistant user's guide version 5.6. INRIA Rocquencourt and ENS Lyon.

[10] J.-P. Jouannaud and M. Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349-391, 1997.

[11] J.-P. Jouannaud and A. Rubio. Rewrite orderings for higher-order terms in η-long β-normal form and the recursive path ordering. *Theoretical Computer Science*, 208(1–2):3–31, 1998.

[12] J.-P. Jouannaud and A. Rubio. A Recursive Path Ordering for Higher-Order Terms in η-Long β-Normal Form, 1999.

[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*, 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] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192(1):3-29, 1998.

[16] M. Okada and G. Takeuti. On the theory of quasi ordinal diagrams. In S. G. Simpson, editor, *Logic and Combinatorics*. American Mathematical Society, 1986.

[17] L. C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*. Academic Press, 1990.

[18] F. Pfenning. Elf: A language for logic defi nition and verifi ed meta-programming. In *Proc. 4th IEEE Symp. Logic in Computer Science*, pages 313–322, 1989.

[19] A. Rubio. A fully syntactic AC-RPO. In *Tenth Int. Conf. on Rewriting Techniques and Applications*. Trento, 1999. To appear in LNCS, Springer-Verlag.