Higher-Order Recursive Path Orderings à la carte

Jean-Pierre Jouannaud
LRI, Université de Paris Sud
91405 Orsay, FRANCE
and LIX, Ecole Polytechnique
91400 Palaiseau, FRANCE

Albert Rubio
Technical University of Catalonia
Pau Gargallo 5
08028 Barcelona, SPAIN
Email: rubio@lsi.upc.es

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. Our ambition is to develop for the higher-order/type 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 [4].

At LICS'99, we contributed to this program with 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.

A weakness of this proposal was that terms could only be compared when they had the same functional structure. We remedy to this situation here by first comparing types before to compare the terms themselves. We also consider a richer type structure with type constructors. Finally, we have a richer notion of status allowing to "neutralize" terms of an arrow type that can never be applied along a derivation.

Our ordering operates on arbitrary higher-order terms, therefore applies on higher-order rewriting based on plain pattern matching. On the other hand, because our ordering includes β-reduction, it can be adapted to operate on terms in η-long β-normal form, hence applying to the higher-order rewriting "à la Nipkow" [10], and yielding an ordering much stronger than the already existing ones [9, 7], to the exception of [3], which needs an important user interaction.

To hint at the strength of the ordering described in the present paper, let us mention that almost all examples that we have found in the literature can be easily oriented. Let us finally mention that our ordering is polymorphic, in the sense that it can prove with a single comparison the termination property of all monomorphic instances of a polymorphic rewrite rule.

We refer to [5] and [8] for missing basic notions and notations.

2 Preliminaries

2.1 Types

Given a set $\mathcal{S}$ of *sort symbols* of a fixed arity, denoted by $s : *^n \to *$, 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 $\to$ for *functional types*:

$$\mathcal{T}_{\mathcal{S}^\forall} := s(\mathcal{T}^n_{\mathcal{S}^\forall}) \mid \alpha \mid (\mathcal{T}_{\mathcal{S}^\forall} \to \mathcal{T}_{\mathcal{S}^\forall})$$
$$\text{for } s : *^n \to * \ \in \mathcal{S} \text{ and } \alpha \in \mathcal{S}^\forall$$

Non-variable types are *functional* when headed by the $\to$ symbol, and *data types* otherwise.

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 $\sigma_1 \in \mathcal{T}_{\mathcal{S}^\forall}, \ldots, \sigma_n \in \mathcal{T}_{\mathcal{S}^\forall}$, and an output type $\sigma \in \mathcal{T}_{\mathcal{S}^\forall}$ such that $\mathcal{V}ar(\sigma) \subseteq \bigcup_i \mathcal{V}ar(\sigma_i)$, written $f : \sigma_1 \times \ldots \times \sigma_n \to \sigma$ and called a *type declaration*. A type declaration is *first-order* if it uses only sorts, and higher-order otherwise; *polymorphic* if it uses some non-ground type, and *monomorphic* otherwise. Given a *type substitution* ξ, we denote by $\mathcal{F}\xi$ the signature $\{f : \sigma_1\xi \times \ldots \times \sigma_n\xi \to \sigma\xi \mid f : \sigma_1 \times \ldots \times \sigma_n \to \sigma \in \mathcal{F}\}$. Polymorphic signatures capture infinitely many monomorphic ones via type instantiation.

2.3 Terms

The set of *raw algebraic λ-terms* is generated from the signature $\mathcal{F}$ and a denumerable set $\mathcal{X}$ of variables according to the grammar rules:

$$\mathcal{T} := \mathcal{X} \mid (\lambda\mathcal{X} : \mathcal{T}_{\mathcal{S}^\forall}.\mathcal{T}) \mid @(\mathcal{T}, \mathcal{T}) \mid \mathcal{F}(\mathcal{T}, \ldots, \mathcal{T}).$$

As a matter ofg convenience, we will often omit types as well as the application operator, writing $u(v)$ for $@(u, v)$ and $u(v_1, \ldots, v_n)$ or $@(u, v_1, \ldots, v_n)$ for $u(v_1) \ldots (v_n)$. As usual, we do not distinguish α-convertible terms.

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

$$\frac{\begin{array}{c} F : \sigma_1 \times \ldots \times \sigma_n \to \sigma \in \mathcal{F} \\ \xi \text{ some type instantiation} \\ \Gamma \vdash t_1 : \sigma_1\xi \ \ldots \ \Gamma \vdash t_n : \sigma_n\xi \end{array}}{\Gamma \vdash F(t_1, \ldots, t_n) : \sigma\xi}$$

Abstraction

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

Application

$$\frac{\Gamma \vdash s : \sigma \to \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash @(s, t) : \tau}$$

(Higher-order) substitutions are written as in $\{x_1 : \sigma_1 \mapsto (\Gamma_1, t_1), \ldots, x_n : \sigma_n \mapsto (\Gamma_n, t_n)\}$ where, for every $i \in [1..n]$, (i) Γ_i is an environment such that $\Gamma_i \vdash t_i : \sigma_i$, and (ii) t_i is assumed different from x_i. We will often omit both the type σ_i and the environment Γ_i in $x_1 : \sigma_1 \mapsto (\Gamma_1, t_1)$. We use the letter γ for substitutions and postfix notation for their application.

2.5 Higher-order rewrite rules

A (possibly higher-order) *term rewriting system* is a set of rewrite rules $R = \{\Gamma_i \vdash l_i \to r_i\}_i$, 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 $\Gamma \vdash l \to r$ 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.

Example 1 We give here the specification for Gdel's system T. Let $\mathcal{S} = \{\mathbb{N}\}$, $\mathcal{S}^\forall = \{\alpha\}$, $\mathcal{F} = \{0 : \mathbb{N},\ s : \mathbb{N} \to \mathbb{N},\ rec : \mathbb{N} \times \alpha \times (\mathbb{N} \to \alpha \to \alpha) \to \alpha\}$, and $\mathcal{X} = \{x : \mathbb{N},\ U : \alpha,\ X : \mathbb{N} \to \alpha \to \alpha\}$. Gödel's recursor for natural numbers is defined by the following rewrite rules:

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

Due to the use of polymorphic signatures, there is only one recursor rule, instead of inifinitely many rules described by one rule schema as in the original presentation. □

2.6 Polymorphic Higher-Order Reduction Orderings

Definition 2.1 *A higher-order reduction ordering is a well-founded, monotonic and stable ordering $>$ of the set of higher-order terms, such that $\longrightarrow_\beta\ \subseteq\ >$. It is in addition* polymorphic *if $s > t$ in a polymorphic signature $\mathcal{F}$ implies $s > t$ in all monomorphic instances $\mathcal{F}\xi$ of $\mathcal{F}$.*

Property 2.2 *Let $>$ be a higher-order reduction ordering and $R = \{\Gamma_i \vdash l_i \to r_i\}_{i \in I}$ be a higher-order rewrite system such that $l_i > r_i$ for every $i \in I$. Assume further that $>$ is polymorphic if so is R. Then the relation $\longrightarrow_{R\xi} \cup \longrightarrow_\beta$ is strongly normalizing in $\mathcal{T}(\mathcal{F}\xi, \mathcal{X})$ for all type instantiations ξ.*

We now turn to normalized higher-order rewriting, which allows to define computations over λ-terms in β-normal η-expanded form. In this context, we consider that function symbols have a basic output, ruling out polymorphism. We will use the notations $\longrightarrow_{R^\eta_\beta}$ for normalized rewriting with R.

Definition 2.3 *A subrelation $>^{\eta}_{\beta}$ of $>$ issaid to be β-stable if $s >^{\eta}_{\beta} t$ implies $s\gamma \downarrow_\beta > t\gamma \downarrow_\beta$ for all normalized terms s, t and normalized substitutions γ.*

Property 2.4 *Let $>$ be a higher-order reduction ordering with a β-stable subrelation $>^{\eta}_{\beta}$, and let $R = \{\Gamma_i \vdash l_i \rightarrow r_i\}_{i \in I}$ be a higher-order rewrite system such that $l_i >^{\eta}_{\beta} r_i$ for every $i \in I$. Then the relation $\longrightarrow_{R^{\eta}_{\beta}}$ is strongly normalizing.*

3 The Higher-Order Recursive Path Ordering

In this section, we present a first version of our ordering, together with examples of its use.

3.1 The HORPO ordering

Our ordering on typed higher-order terms uses three basic ingredients, presented in the following three subsections. The ordering itself is defined the fourth subsection.

3.1.1 Type ordering

We assume given two quasi-ordering on types $\geq_{\mathcal{T}_S} \subseteq \geq^{\rightarrow}_{\mathcal{T}_S}$ which are assumed to satisfy the following properties:

1. Well-foundedness: $>^{\rightarrow}_{\mathcal{T}_S}$ is well-founded;
2. arrow preservation: $\tau \rightarrow \sigma =_{\mathcal{T}_S} \alpha$ implies $\alpha = \tau' \rightarrow \sigma'$, $\tau' =_{\mathcal{T}_S} \tau$ and $\sigma =_{\mathcal{T}_S} \sigma'$;
3. arrow decrease: $\tau \rightarrow \sigma >_{\mathcal{T}_S} \alpha$ implies $\sigma \geq_{\mathcal{T}_S} \alpha$ or $\alpha = \tau' \rightarrow \sigma'$, $\tau' =_{\mathcal{T}_S} \tau$ and $\sigma >_{\mathcal{T}_S} \sigma'$;
4. Stability under type substitution: If $\sigma >_{\mathcal{T}_S} \tau$, ($\sigma =_{\mathcal{T}_S} \tau$, respectively) then $\sigma\xi >_{\mathcal{T}_S} \tau\xi$ for every ground type substitution ξ ($\sigma\xi =_{\mathcal{T}_S} \tau\xi$, respectively);
5. Compatibility of signature declarations with type instantiations: for any function symbol $f\sigma_1 \times \sigma_n \rightarrow \sigma$ and type instantiations ξ', ξ'' such that $\sigma_i\xi' =_{\mathcal{T}_S} \sigma_i\xi''$, then $\sigma\xi' =_{\mathcal{T}_S} \sigma\xi''$.

Notation: We denote by $\mathcal{T}_S^{small}$ the non-empty set of ground data types which are not bigger than an arrow type in the ordering $\geq_{\mathcal{T}_S}$.

3.1.2 Statuses and Precedence

We assume given a partition $Mul \uplus Lex$ of $\mathcal{F}$ and a well-founded *precedence* $\geq_{\mathcal{F}}$ on $\mathcal{F}$ such that
- $f : \overline{\sigma} \rightarrow \sigma \geq_{\mathcal{F}} g : \overline{\tau} \rightarrow \tau$ if $\sigma >_{\mathcal{T}_S} \tau$;
- if $f : \overline{\sigma} \rightarrow \sigma =_{\mathcal{F}} g : \overline{\tau} \rightarrow \tau$, then $f \in Lex$ iff $g \in Lex$ and $\overline{\sigma} =_{\mathcal{T}_S} \overline{\tau}$;
- variables are incomparable in $>_{\mathcal{F}}$ among themselves and with other function symbols;

3.1.3 Definition of the ordering

We can now give our definition of the higher-order recursive path ordering, which builds upon Dershowitz's recursive path ordering for first-order terms [4], and improves significantly over [8]. Apart from the two new cases allowing to deal with applications on the right or left, and the monotonicity rules for application and abstraction, the main difference with the first order recursive path ordering is that we take care of higher-order terms in the arguments of the smaller side by having a corresponding bigger higher-order term in the arguments of the bigger side. This idea is captured in the following proposition:

$$A = \forall v \in \overline{t}\ s \underset{horpo}{\succ} v \text{ or } u \underset{horpo}{\succeq} v \text{ for some } u \in \overline{(}s)$$

Definition 3.1 $s : \sigma \succ_{horpo} t : \tau$ *iff* $\sigma \geq_{\mathcal{T}_S} \tau$ *and*

1. $s = f(\overline{s})$ *with* $f \in \mathcal{F}$, *and* $u \succeq_{horpo} t$ *for some* $u \in \overline{s}$
2. $s = f(\overline{s})$ *and* $t = g(\overline{t})$ *with* $f >_{\mathcal{F}} g$, *and* A
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* A
5. $s = @(s_1, s_2)$ *and* $s_1 \succeq_{horpo} t$ *or* $s_2 \succeq_{horpo} t$
6. $s = \lambda x : \alpha.u$ *with* $x \notin \mathcal{V}ar(t)$ *and* $u \succeq_{horpo} t$
7. $f \in \mathcal{F}$, $@(\overline{t})$ *is some partial left-flattening of* t, *and* A
8. $s = f(\overline{s})$ *with* $f \in \mathcal{F}$, $t = \lambda x : \alpha.v$ *with* $x \notin \mathcal{V}ar(v)$ *and* $s \succ_{horpo} v$
9. $s = @(s_1, s_2)$, $t = @(\overline{t})$ *is a partial left-flattening of* t *and* $\{s_1, s_2\}(\succ_{horpo})_{mul}\overline{t}$
10. $s = \lambda x.u$, $t = \lambda x.v$ *and* $u \succ_{horpo} v$
11. $s = @(\lambda x.u, v)$ *and* $u\{x \mapsto v\} \succeq_{horpo} t$

The definition is recursive, and, apart from case 11, recursive calls operate on pairs of terms which are subterms of the starting pair. This ensures the well-foundedness of the definition, since β-reductions are compatible with subterm, by comparing pairs of arguments in the well-founded ordering $(\longrightarrow_\beta \cup \rhd, \rhd)_{lex}$.

Important observations are the following:

Comparing two terms s and t requires comparing their types beforehand. Doing otherwise would not garantee the typability of t in general. Note that the present ordering compares terms of related types, improving over [8], where terms of equivalent types only could be compared (the equivalence being generated by the equality on sorts).

In Case 1, the redundancy anticipates over Section 4, in which $\overline{s}$ and $\mathcal{CC}(s)$ are different sets.

When the signature is first-order, cases 1, 2, 3, and 4 together reduce to the usual recursive path ordering for first-order terms, whose complexity is known to be in $O(n^2)$ for an input of size n.

Example 2 (example 1 continued) We use here for the ordering on types the equivalence generated by the equality of the sort constants. The first rule succeeds immediately by case 1. For the second rule, we apply case 7, 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 example was already proved in [8]. 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 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 3 This example is partly taken from [8]. 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:

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

Using the same ordering on types as previously, 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 7, 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 $List$.

Using now the richer type discipline described in the present paper, we can reformulate this example by using a sort constructor $List : * \rightarrow *$. Assuming $List(\xi) >_{\mathcal{T}_S} \xi$ in our ordering, the computation goes exactly as previously. □

We finally state the main result of this section.

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

3.2 Examples

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

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

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

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

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 9.

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

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.

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

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. $\Box$

This example is more tricky, since we need to build transitivity:

Example 5 Surjective disjoint union : let $\mathcal{S} = \{A, B, U\}$, $\alpha \in \{A, B, U\}$, $\mathcal{F} = \{inl : A \rightarrow U;\ inr : B \rightarrow U;\ case_\alpha : U \times (A \rightarrow \alpha) \times (B \rightarrow \alpha) \rightarrow \alpha\}$. The rules are:

$$\begin{array}{rcl} case_\alpha(inl(X), F, G) & \rightarrow & @(F, X) \\ case_\alpha(inr(Y), F, G) & \rightarrow & @(G, Y) \\ case_\alpha(Z, \lambda x.H(inl(x)), \lambda y.H(inr(y))) & \rightarrow & @(H, Z) \end{array}$$

The first two rules are by case 7. For the last, we show that $case_\alpha(Z, \lambda x.H(inl(x)), \lambda y.H(inr(y))) \succ_{horpo} @(\lambda x.H(x), Z) \succ_{horpo} @(H, Z)$. The first comparison is by Case 7, generating the proof obligation $\lambda x.H(inl(x) \succ_{horpo} \lambda x.H(x)$ which succeeds by first Case 10 then Case 1. The second is by Case 11.

3.3 A uniform ordering on terms and types

The idea is to use a recursive path ordering on the type structure for $\geq_{\mathcal{T}_S}^{\rightarrow}$ with an appropriate restriction for $\geq_{\mathcal{T}_S}$, therefore allowing us to use the same ordering structure for both terms and types. This requires adding the new constant $*$ in our language for typing types. We omit the corresponding type system which is obvious. We now assume given

- a partition $Mul \uplus Lex$ of $\mathcal{F} \cup \mathcal{S} \cup \{@\}$ such that $@ \in Mul$ and all function symbols are compatible;

-a well-founded ordering $\geq_{\mathcal{FS}}$ on $\mathcal{F} \cup \mathcal{S}$, called the precedence such that variables are incomparable among themselves and with other symbols.

Definition 3.3 *Let* $A = \forall v \in \overline{t}\ s \succ_{horpo} v$ *or* $u \succeq_{horpo} v$ *for some* $u \in \overline{s}$
Then, $s : \sigma \succ_{horpo} t : \tau$ *iff* $\sigma = \tau = *$ *or* $\sigma \geq_{\mathcal{T}_S} \tau$ *and*

1. $s = f(\overline{s})$ *with* $f \in \mathcal{FS}$, *and* $u \succeq_{horpo} t$ *for some* $u \in \overline{s}$
2. $s = f(\overline{s})$ *and* $t = g(\overline{t})$ *with* $f >_{\mathcal{FS}} g$, *and* A
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* A
5. $s = @(s_1, s_2)$ *and* $s_1 \succeq_{horpo} t$ *or* $s_2 \succeq_{horpo} t$
6. $s = \lambda x : \alpha.u$ *with* $x \notin \mathcal{V}ar(t)$ *and* $u \succeq_{horpo} t$
7. $f \in \mathcal{F}$, $@(\overline{t})$ *is some partial left-flattening of* t, *and* A
8. $s = f(\overline{s})$ *with* $f \in \mathcal{F}$, $t = \lambda x : \alpha.v$ *with* $x \notin \mathcal{V}ar(v)$ *and* $s \succ_{horpo} v$
9. $s = @(s_1, s_2)$, $t = @(\overline{t})$ *is a partial left-flattening of* t *and* $\{s_1, s_2\}(\succ_{horpo})_{mul}\overline{t}$
10. $s = \lambda x.u$, $t = \lambda x.v$ *and* $u \succ_{horpo} v$
11. $s = @(\lambda x.u, v)$ *and* $u\{x \mapsto v\} \succeq_{horpo} t$
12. $s = \alpha \rightarrow \beta$ *and* $\beta \succeq_{horpo} t$
13. $s = \alpha \rightarrow \beta,\ t = \alpha' \rightarrow \beta', \alpha = \alpha'$ *and* $\beta \succ_{horpo} \beta'$

This uniform definition opens the way for defining a recursive path ordering for terms in a dependent type discipline, and therefore, for the full calculus of constructions. The difficulty should not be underestimated, though, since it will allow us to prove strong normalization of this calculus augmented with rules at the type level, as it is done in [2].

4 Computational Closure

The ordering is quite sensitive to innocent variations of the language, like adding (higher-order) dummy arguments to righthand sides. We will now solve these problems by incorporating a new mecanism adapted from [1] to the ordering which encompasses the computability arguments developped by Girard to show the strong normalization of System F. First, we need introducing a restriction of the subterm ordering:

$s : \sigma \rhd_{>_{\mathcal{T}_S}} t : \tau$ iff $\sigma \geq_{\mathcal{T}_S} \tau$ and $s \rhd \tau$.

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* $\overline{t}$*, and closed under the following operations:*

1. *subterm: let $s \in \mathcal{CC}(t, \mathcal{V})$ and $u : \sigma$ be a subterm of s such that $\sigma \in \mathcal{T}_S^{small}$ and $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$; then $u \in \mathcal{CC}(t, \mathcal{V})$;*

2. *precedence: let g such that $f >_{\mathcal{F}} g$, and $\overline{s} \in \mathcal{CC}(t, \mathcal{V})$; then $g(\overline{s}) \in \mathcal{CC}(t, \mathcal{V})$;*

3. *recursive call: let $\overline{s}$ be a sequence of terms in $\mathcal{CC}(t, \mathcal{V})$ such that $f(\overline{s})$ is well typed and $\overline{t}(\succ_{horpo} \cup \rhd_{>_{\mathcal{T}_S}})_{stat_f}\overline{s}$; then $f(\overline{s}) \in \mathcal{CC}(t, \mathcal{V})$;*

4. *application: let $s : \sigma_1 \to \ldots \to \sigma_n \to \sigma \in \mathcal{CC}(t, \mathcal{V})$ and $u_i : \sigma_i \in \mathcal{CC}(t, \mathcal{V})$ for every $i \in [1..n]$; then $@(s, u_1, \ldots, u_n) \in \mathcal{CC}(t, \mathcal{V})$;*

5. *abstraction: let $x \notin \mathcal{V}ar(t) \cup \mathcal{V}$ and $s \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$; then $\lambda x.s \in \mathcal{CC}(t, \mathcal{V})$;*

6. *reduction: let $u \in \mathcal{CC}(t, \mathcal{V})$, and $u \succeq_{horpo} v$; then $v \in \mathcal{CC}(t, \mathcal{V})$;*

7. *weakening: let $x \notin \mathcal{V}ar(u, t) \cup \mathcal{V}$. Then, $u \in \mathcal{CC}(t, \mathcal{V})$ iff $u \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$.*

As an exercise, the reader may want to prove that, given s, u, x, V such that $\mathcal{V}ar(u) \subseteq V$ and $x \notin V$, then $u \in \mathcal{CC}(s, V)$ iff $\lambda x.@(u, x) \in \mathcal{CC}(s, V)$. The only if part uses successively Cases 7, 4 together with the basic case and 5. The if part is more tricky, but there is a proof using Case 6 twice only.

Note that we use recursively the ordering of the previous section in order to defined the recursive case of the closure, therefore giving extra power to the closure. On the other hand, the subterm relationship had to be restricted to terms of decreasing types in order to ensure the well-foundedness of the union (via a commutation principle, $\succ_{horpo}$ being monotonic on terms of equivalent types).

Because of rule 6, 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.

We can now modify our ordering as follows:

Definition 4.2 *Let* $A = \forall v \in \overline{t}\ s \succ_{horpo} v$ *or* $u \succeq_{horpo} v$ *for some* $u \in \mathcal{CC}(s)$.
Then, $s : \sigma \succ_{horpo} t : \tau$ *iff* $\sigma \geq_{\mathcal{T}_S} \tau$ *and*

1. *$s = f(\overline{s})$ with $f \in \mathcal{F}$, and $u \succeq_{horpo} t$ for some $u \in \overline{s}$ or $t \in \mathcal{CC}(s)$*

2. *$s = f(\overline{s})$ and $t = g(\overline{t})$ with $f >_{\mathcal{F}} g$, and A*

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* A
5. $s = @(s_1, s_2)$ *and* $s_1 \succeq_{horpo} t$ *or* $s_2 \succeq_{horpo} t$
6. $s = \lambda x : \alpha.u$ *with* $x \notin \mathcal{V}ar(t)$ *and* $u \succeq_{horpo} t$
7. $f \in \mathcal{F}$, $@(\overline{t})$ *is some partial left-flattening of* t, *and* A
8. $s = f(\overline{s})$ *with* $f \in \mathcal{F}$, $t = \lambda x : \alpha.v$ *with* $x \notin \mathcal{V}ar(v)$ *and* $s \succ_{horpo} v$
9. $s = @(s_1, s_2)$, $t = @(\overline{t})$ *is a partial left-flattening of* t *and* $\{s_1, s_2\}(\succ_{horpo})_{mul}\overline{t}$
10. $s = \lambda x.u$, $t = \lambda x.v$ *and* $u \succ_{horpo} v$
11. $s = @(\lambda x.u, v)$ *and* $u\{x \mapsto v\} \succeq_{horpo} t$

The definition is recursive, and we can show that it is well-founded by comparing pairs of terms in the well-founded ordering $(\succ_{horpo}, \rhd)_{right-to-left-lex}$. As previously, we have:

Theorem 4.3 $\succ_{horpo}$ *is included in a polymorphic higher-order 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 6 $\quad n * m \;\; \rightarrow \;\; rec(n, 0, \lambda z_1 z_2.m + z_2)$
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 2 of the definition of the computational closure. Applying case 5 twice yields then the result. □

Example 7 Let $\mathcal{S} = \{form\}$, $\mathcal{F} = \{D : (form \rightarrow form) \rightarrow (form \rightarrow form);\ \forall, \exists : (form \rightarrow form) \rightarrow form;\ 0, 1 : form; +, \times : form \times form \rightarrow form\}$. The rules are:

$$\begin{array}{rcl} D(\lambda x.y & \rightarrow & \lambda x.0 \\ D(\lambda x.x & \rightarrow & \lambda x.1 \\ D(\lambda x.(F\ x) + (G\ x)) & \rightarrow & \lambda x.(D(F)x) + (D(G)x) \\ D(\lambda x.(F\ x) \times (G\ x)) & \rightarrow & \lambda x.(D(F)x) \times (G\ x) + (F\ x) \times (D(G)x) \end{array}$$

We take $D >_{\mathcal{F}} \{0, 1, x, +\}$ for precedence and assume all statuses are multiset. The interesting rule is the derivation of products, which is done entirely by the mechanism of the closure. We will take this opportunity to make a goal directed proof, using a stack of subgoals:
Initial goal (for sake of clarity, we rename the bound variables):
$\lambda y.(D(F)y) \times (G\ y) + (F\ y) \times (D(G)y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)))$
By abstraction:
$(D(F)y) \times (G\ y) + (F\ y) \times (D(G)y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
By precedence, we obtain two subgoals:
$(D(F)y) \times (G\ y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$(F\ y) \times (D(G)y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$

By precedence again, we get two new subgoals, resulting in a total of three:
$(D(F)y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$(G\ y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$(F\ y) \times (D(G)y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
By application:
$D(F) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$y \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$(G\ y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$(F\ y) \times (D(G)y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
By recursive call (note that $\lambda x.(F\ x) \times (G\ x)) \rhd_{>_{\mathcal{T}_S}} F$)
$F \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$y \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$(G\ y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$(F\ y) \times (D(G)y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
By reduction since $\lambda x.(F\ x) \succ_{horpo} F$ by Case 6:
$\lambda x.(F\ x) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$y \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$(G\ y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$(F\ y) \times (D(G)y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
By basic case:
$y \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$(G\ y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$(F\ y) \times (D(G)y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
By basic case again:
$(G\ y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$(F\ y) \times (D(G)y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
By application:
$G \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$y \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
$(F\ y) \times (D(G)y) \in \mathcal{CC}(D(\lambda x.(F\ x) \times (G\ x)), \{y\})$
The first subgoal disappears by using, as previously, Case 6 and basic case successively, while the second is solved by basic case. The third is then done in the same way as the previous computation.

We now come to an interesting example, Currying, for which we can show that our ordering allows us to *chose* the amount of Currying we like.

Example 8 First, to a signature $\mathcal{F} = \{\ldots, f : \sigma_1 \times \ldots \times \sigma_n \rightarrow (\sigma_{n+1} \rightarrow \ldots \sigma_p \rightarrow \sigma), \ldots\}$, we associate the signature $\mathcal{F}^{Curry} = \{\ldots, f_{i\in[1..p]} : \sigma_1 \times \ldots \times \sigma_i \rightarrow (\sigma_{i+1} \rightarrow \ldots \sigma_p \rightarrow \sigma), \ldots\}$, and we give the following rewrite rules:

$$\begin{array}{rcl} f_{j+1}(t_1, \ldots, t_{j+1}) & \rightarrow & @(f_{j+1}(t_1, \ldots, t_j), t_{j+1}) \\ @(f_i(t_1, \ldots, t_i), t_{i+1}) & \rightarrow & f_{i+1}(t_1, \ldots, t_{i+1}) \end{array}$$

Setting now $f_{j+1} >_{\mathcal{T}_S} f_j$, we show easily that the righthand side of the first rule is in the closure of its lefthand side. Setting now $f_i >_{\mathcal{T}_S} f_{i+1}$, we can do the second comparison. This is a little bit more delicate, and we must use the middle term trick, with $@(\lambda x.f_{i+1}(t_1, \ldots, t_i, x), t_{i+1})$. The details are left to the reader.

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

Example 9 (taken from [3]) 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. □

5 Neutralizing abstractions

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 9 shows that this is not always the case. In this example, the righthand side arrow type subterm $\lambda y.P(y) \cdot x$ does not receive a special treatment, although it cannot serve creating a redex in a derivation. We will now improve our ordering by introducing a special treatment for these terms. The idea is to equip each function symbols with a set of neutralized positions, which can be seen as an additional status for that purpose.

Notation: In this section, we assume given two sets of variables: the set $\mathcal{X}$ and a set $\mathcal{X}_\mathcal{T}$ of variables disjoint from $\mathcal{X}$, s.t. $x_\sigma : \sigma \in \mathcal{X}_\mathcal{T}$ iff $x : \sigma \in X$ and σ is a functional type or a type variable. Note, this is important, that x_σ is a variable different from x. Of course, the higher-ordre rules we want to prove terminating are built from terms in $\mathcal{T}(\mathcal{X})$, not in $\mathcal{T}(\mathcal{X} \cup \mathcal{X}_\mathcal{T})$.

Definition 5.1 *The* neutralization $N(t)$ *of a term* $t \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{X}_\mathcal{T})$ *is defined as:*

1. *if t is of the form $\lambda x : \sigma.u$ then $N(t) = N(u\{x \mapsto \bot_\sigma\})$, where $\bot_\sigma : \sigma$ is a constant added to the signature $\mathcal{F}_\sigma$;*

2. *if t is the variable $y : \sigma \in \mathcal{X}$, where σ is an functional type or a type variable then $N(t) = y_\sigma$.*

3. *Otherwise $N(t) = t$.*

We are now ready for improving our higher-order recursive path ordering once more.

Definition 5.2 *To each symbol $f \in \mathcal{F}$, we associate a (possibly empty) set $\mathcal{N}_f \subseteq [1..ar(f)]$ of* neutralized *positions. To each term t of the form $f(t_1, \ldots, t_n)$, we associate the list $\mathcal{NA}(t)$ of* neutralized arguments *of f in t, defined as the list $t'_1, \ldots, t'_n$, where t'_i is $N(t_i)$ if $i \in \mathcal{N}_f$ and t_i otherwise.*

The computable closure needs adaptation:

Definition 5.3 *Given a term $t = f(\overline{t})$, we define its neutralized* computable closure *$\mathcal{CC}(t)$ as $\mathcal{CC}(t, \emptyset)$, where $\mathcal{CC}(t, \mathcal{V})$, with $\mathcal{V} \subseteq \mathcal{X} \setminus \mathcal{V}ar(t)$, is the smallest set of well-typed terms containing all variables in $\mathcal{V}$, all constants $\perp_\sigma$ for $\sigma \in \mathcal{T}_{\mathcal{S}^\forall}$ and all terms in $\mathcal{NA}(t)$, and closed under the following operations:*

1. *$x_\sigma \in \mathcal{CC}(t, \mathcal{V})$ iff $x \in \mathcal{CC}(t, \mathcal{V}$ when σ is not a data type;*
2. *subterm of minimal type: let $s \in \mathcal{CC}(t, \mathcal{V})$ and $u : \sigma$ be a subterm of s such that $\sigma \in \mathcal{T}_{\mathcal{S}}^{small}$ and $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$; then $u \in \mathcal{CC}(t, \mathcal{V})$;*
3. *precedence: let u be $g(\overline{u})$ with $f >_{\mathcal{F}} g$, and $\mathcal{NA}(u) \in \mathcal{CC}(t, \mathcal{V})$; then $u \in \mathcal{CC}(t, \mathcal{V})$;*
4. *recursive call: let u be $f(\overline{u})$, with $\mathcal{NA}(u) \in \mathcal{CC}(t, \mathcal{V})$ and $\mathcal{NA}(t)(\succ_{horpo})_{stat_f}\mathcal{NA}(u)$; then $u \in \mathcal{CC}(t, \mathcal{V})$;*
5. *application: let $s : \sigma_1 \to \ldots \to \sigma_n \to \sigma \in \mathcal{CC}(t, \mathcal{V})$ and $u_i : \sigma_i \in \mathcal{CC}(t, \mathcal{V})$ for every $i \in [1..n]$; then $@(s, u_1, \ldots, u_n) \in \mathcal{CC}(t, \mathcal{V})$;*
6. *abstraction: let $x \notin \mathcal{V}ar(t) \cup \mathcal{V}$ and $s \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$; then $\lambda x.s \in \mathcal{CC}(t, \mathcal{V})$;*
7. *reduction: let $u \in \mathcal{CC}(t, \mathcal{V})$, and $u \longrightarrow_\beta v$; then $v \in \mathcal{CC}(t, \mathcal{V})$.*
8. *extensionality: let $u = \lambda x.@(v, x) \in \mathcal{CC}(t, \mathcal{V})$ with $x \notin \mathcal{V}ar(v)$; then $v \in \mathcal{CC}(t, \mathcal{V})$.*

Given a status and a set of neutralized positions $\mathcal{N}_f$ for every algebraic function symbol f, the ordering is defined as follows:

Definition 5.4 *Then $s : \sigma \succ_{horpo} t : \tau$ iff $\sigma \geq_{\mathcal{T}_\mathcal{S}} \tau$ and*

1. *$s = f(\overline{s})$ with $f \in \mathcal{F}$, and*
   (i) $s_i \succeq_{horpo} t$ for some $i \notin \mathcal{N}_f$ or (ii) $\mathcal{N}(u) \succeq_{horpo} t$ for some $u \in \overline{s}$ or $t \in \mathcal{CC}(s)$
2. *$s = f(\overline{s})$ and $t = g(\overline{t})$ with $f >_{\mathcal{F}} g$, and A*
3. *$f = g \in Mul$ and $\mathcal{NA}(s)(\succ_{horpo})_{mul}\mathcal{NA}(t)$*
4. *$f = g \in Lex$ and $\mathcal{NA}(s)(\succ_{horpo})_{lex}\mathcal{NA}(t)$ and A*
5. *$s = @(s_1, s_2)$ and $s_1 \succeq_{horpo} t$ or $s_2 \succeq_{horpo} t$*
6. *$s = \lambda x : \alpha.u$ with $x \notin \mathcal{V}ar(t)$ and $u \succeq_{horpo} t$*
7. *$f \in \mathcal{F}$, $@(\overline{t})$ is some partial left-flattening of t, and A*

8. $s = f(\overline{s})$ *with* $f \in \mathcal{F}$, $t = \lambda x : \alpha.v$ *with* $x \notin \mathcal{V}ar(v)$ *and* $s \succ_{horpo} v$

9. $s = @(s_1, s_2)$, $t = @(\overline{t})$ *is a partial left-flattening of* t *and* $\{s_1, s_2\}(\succ_{horpo})_{mul}\overline{t}$

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

11. $s = @(\lambda x.u, v)$ *and* $u\{x \mapsto v\} \succeq_{horpo} t$

where $A = \forall v \in \mathcal{NA}(t)$ $s \succ_{horpo} v$ *or* $u \succeq_{horpo} v$ *for some* $u \in \mathcal{CC}(s)$.

Note that Cases 1(i) and 1(ii) both apply to a subterm which is not at a neutralized position. On the other hand, Cases 1(ii) only applies to subterms at a neutralized position. Again,

Theorem 5.5 *$\succeq_{horpo}$ is included in a polymorphic higher-order reduction ordering.*

We can now prove the example that was left over in the previous section. Assuming that the argument of Σ is neutralized, we verify that the first eight rules are still oriented. The comparisons change a bit for the last three (6,7,8. Let us recall the one left over:

$$\Sigma(\lambda y.P(y)) \cdot x \;\rightarrow\; \Sigma(\lambda y.P(y) \cdot x)$$

Adding $\cdot >_{\mathcal{F}} \Sigma$ to the precedence, we now prove this rule with the following sequence of goals:
Initial goal: $\Sigma(\lambda y.P(y)) \cdot x \succ_{horpo} \Sigma(\lambda y.P(y) \cdot x)$
By Case 2: $\Sigma(\lambda y.P(y)) \cdot x \succ_{horpo} P(\bot) \cdot x$
By Case 4: $\Sigma(\lambda y.P(y)) \cdot x \succ_{horpo} P(\bot)$ and $\Sigma(\lambda y.P(y)) \cdot x \succ_{horpo} x$
Both are done easily now, as in previous similar examples.

6 Normalized rewriting

Example 10 (taken from Nipkow) Let $\mathcal{S} = \{form, ind\}$, $\mathcal{F} = \{\wedge, \vee : form \times form \rightarrow form; \neg : form \rightarrow form; \forall, \exists : (ind \rightarrow form) \rightarrow form\}$. The rules are the following:

$$\begin{array}{rcl}
P \wedge \forall(\lambda x.Q(x)) & \rightarrow & \forall(\lambda x.(P \wedge Q(x))) \\
\forall(\lambda x.Q(x)) \wedge P & \rightarrow & \forall(\lambda x.(Q(x) \wedge P)) \\
P \vee \forall(\lambda x.Q(x)) & \rightarrow & \forall(\lambda x.(P \vee Q(x))) \\
\forall(\lambda x.Q(x)) \vee P & \rightarrow & \forall(\lambda x.(Q(x) \vee P)) \\
P \wedge \exists(\lambda x.Q(x)) & \rightarrow & \exists(\lambda x.(P \wedge Q(x))) \\
\exists(\lambda x.Q(x)) \wedge P & \rightarrow & \exists(\lambda x.(Q(x) \wedge P)) \\
P \vee \exists(\lambda x.Q(x)) & \rightarrow & \exists(\lambda x.(P \vee Q(x))) \\
\exists(\lambda x.Q(x)) \vee P & \rightarrow & \exists(\lambda x.(Q(x) \vee P)) \\
\neg(\forall(\lambda x.Q(x))) & \rightarrow & \exists(\lambda x.\neg(Q(x))) \\
\neg(\exists(\lambda x.Q(x))) & \rightarrow & \forall(\lambda x.\neg(Q(x)))
\end{array}$$

This example makes sense in the context of normalized rewriting only. We now sketch how to restrict the higher-order recursive path ordering from Section 5 in order to obtain a β-stable subrelation which can then be used to show strong normalization of this example, when using higher-order pattern matching "à la Nipkow".

First, the subterm case in the definition of the closure must be restricted, making sure that it is not a subterm of an application of the form $@(x, w)$. Second, terms in the closure need to be normalized.

The rules above are then treated by using an appropriate precedence. The dateils are left to the reader.

7 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 [1]. 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.

References

[1] F. Blanqui, J.-P. Jouannaud, and M. Okada. The Calculus of Algebraic Constructions. In *RTA'99*, 1999.

[2] F. Blanqui. Definitions by rewriting in the calculus of constructions. In *LICS'01*, 2001.

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

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

[5] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–309. North-Holland, 1990.

[6] Jean-Pierre Jouannaud and Mitsuhiro Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349–391, February 1997.

[7] Jean-Pierre Jouannaud and Albert 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, November 1998.

[8] Jean-Pierre Jouannaud and Albert Rubio. The higher-order recursive path ordering. In Giuseppe Longo, editor, *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, Trento, Italy, July 1999. IEEE Comp. Soc. Press.

[9] 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.

[10] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192(1):3–29, February 1998.