A monotonic Higher-Order Semantic Path Ordering

Cristina Borralleras[1] and Albert Rubio[2]

[1] Universitat de Vic, Spain
Email: cristina.borralleras@uvic.es
[2] Universitat Politècnica de Catalunya, Barcelona, SPAIN
Email: rubio@lsi.upc.es

Abstract. There is an increasing use of (first- and higher-order) rewrite rules in many programming languages and logical systems. The *recursive path ordering* (RPO) is a well-known tool for proving termination of such rewrite rules in the first-order case. However, RPO has some weaknesses. For instance, since it is a simplification ordering, it can only handle simply terminating systems. Several techniques have been developed for overcoming these weaknesses of RPO. A very recent such technique is the *monotonic semantic path ordering* (MSPO), a simple and easily automatizable ordering which generalizes other more ad-hoc methods.
Another recent extension of RPO is its higher-order version HORPO. HORPO is an ordering on terms of a typed lambda-calculus generated by a signature of higher-order function symbols. Although many interesting examples can be proved terminating using HORPO, it inherits the weaknesses of the first-order RPO.
Therefore, there is an obvious need for higher-order termination orderings without these weaknesses. Here we define the first such ordering, the *monotonic higher-order semantic path ordering* (MHOSPO), which is still automatizable like MSPO. We give evidence of its power by means of several natural and non-trivial examples which cannot be handled by HORPO.

1 Introduction

There is an increasing use of higher-order rewrite rules in many programming languages and logical systems. As in the first-order case, termination is a fundamental property of most applications of higher-order rewriting. Thus, there exists a need to develop for the higher-order case the kind of semi-automated termination proof techniques that are available for the first-order case.

There have been several attempts at designing methods for proving strong normalization of higher-order rewrite rules based on ordering comparisons. These orderings are either quite weak [LSS92,JR98], or need an important user interaction [PS95]. Recently, in [JR99], the recursive path ordering (RPO) [Der82] —the most popular ordering-based termination proof method for first-order rewriting— has been extended to a higher-order setting by defining a higher-order recursive path ordering (HORPO) on terms following a typing discipline including ML-like polymorphism. This ordering is powerful enough to deal with many non-trivial examples and can be automated. Besides, all aforementioned previous methods operate on terms in η-long β-normal form, hence apply only to the higher-order rewriting "à la Nipkow" [MN98], based on higher-order pattern matching modulo $\beta\eta$. HORPO is the first method which operates on arbitrary higher-order terms, therefore applying to the other kind of rewriting, based on plain pattern matching, where β-reduction is considered as any other rewrite rule. Furthermore, HORPO can operate as well on terms in η-long β-normal form, and hence it provides a termination proof method for both kinds of higher-order rewriting (see also [vR01] for a particular version of HORPO dealing η-long β-normal forms).

However, HORPO inherits the same weaknesses that RPO has in the first-order case. RPO is a *simplification ordering* (a monotonic ordering including the subterm relation), which extends a precedence on function symbols to an ordering on terms. It is simple and easy to use, but unfortunately, it turns out, in many cases, to be a weak termination proving tool. First, there are many term rewrite systems (TRSs) that are terminating but are not contained in any simplification ordering, i.e. they are not *simply terminating*. Second, in many cases the head symbol, the one that is compared with the precedence, does not provide enough information to prove the termination of the TRS. Therefore, since HORPO follows the same structure and the same use of a precedence as in RPO (in fact, it reduces to RPO when restricted to first-order terms), it is easy to expect that similar weaknesses will appear when proving termination of higher-order rewriting.

To avoid these weaknesses, in the first-order case, many different so-called *transformation methods* have been developed. By transforming the TRS into a set of ordering constraints, the *dependency pair* method [AG00] has become a successful general technique for proving termination of (non-simply terminating) TRSs.

As an alternative to transformation methods, more powerful term orderings like the *semantic path ordering* (SPO) ([KL80]) can be used. SPO generalizes RPO by replacing the precedence on function symbols by any (well-founded) underlying (quasi-)ordering involving the whole term and not only its head symbol. Although the simplicity of the presentation is kept, this makes the ordering much more powerful. Unfortunately, SPO is not so useful in practice, since, although it is well-founded, it is not, in general, monotonic. Hence, in order to ensure termination, apart from checking that the rules of the rewrite system are included in the ordering, in addition the monotonicity for contexts of the rewrite rules has to be proved.

In [BFR00], a monotonic version of SPO, called MSPO, has been presented. MSPO overcomes the weaknesses of RPO, it is automatable and it is shown to generalize other existing transformation methods.

Due to the fact that RPO and SPO share the same "path ordering nature", our aim is to obtain for SPO and MSPO the same kind of extensions to the higher-order case as it was done for RPO.

In this paper we present the *higher-order semantic path ordering* (HOSPO) which operates on terms of a typed lambda-calculus generated by a signature of higher-order function symbols. As done for HORPO in [JR99], HOSPO is proved well-founded by Tait and Girard's computability predicate proof technique. Then a monotonic version of HOSPO, called MHOSPO, is obtained, which provides an automatable powerful method for proving termination of higher-order rewriting on arbitrary higher-order terms union β-reduction. To illustrate this power several non-trivial examples are shown to be terminating.

In this work we do not consider η-reductions, although all results can be extended to include them at the expense of some easy, but technical, complications. For the same reason we have not included ML-like polymorphism.

Besides its own interest as a termination method for higher-order rewriting, the extension of HORPO to HOSPO and the definition of MHOSPO on top of HOSPO is also interesting for other reasons. On the one hand, it shows the stability of the definition of HORPO, since it extends to HOSPO in the same way as RPO extends to SPO. On the other hand, it shows the stability of the definition of MSPO, since MHOSPO is obtained from HOSPO in the same way as MSPO is obtained from SPO. This gives some intuition of why term orderings provide a more adequate framework for defining general termination proving methods than other techniques.

Formal definitions and basic tools are introduced in Section 2. In Section 3 an example motivating the need of extending HORPO is given. In Section 4 we present and study the higher-order semantic path ordering. Section 5 introduces MHOSPO. The method is applied to two examples in Section 6. In Section 7 an improved version of the computable closure is given. Some conclusions and possible extensions are given in Section 9. The reader is expected to be familiar with the basics of term rewrite systems [DJ90] and typed lambda calculi [Bar92].

2 Preliminaries

2.1 Types, Signatures and Terms

We consider terms of a simply typed lambda-calculus generated by a signature of higher-order function symbols.

The set of types $\mathbb{T}$ is generated from the set $V_{\mathbb{T}}$ of *type variables* (considered as sorts) by the constructor $\rightarrow$ for *functional types* in the usual way. Types are called *functional* when they are headed by the $\rightarrow$ symbol, and *basic* when they are a type variable. As usual, $\rightarrow$ associates to the right. In the following, we use α, β for type variables and $\sigma, \tau, \rho, \theta$ for arbitrary types.

Let $\equiv$ be the congruence on types generated by equating all type variables in $V_{\mathbb{T}}$. Note that two types are equivalent iff they have the same arrow skeleton, i.e. $\sigma \equiv \tau$ iff replacing all type variables in σ and τ by the same type variable α we obtain two identical types.

A signature $\mathcal{F}$ is 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 $\sigma_1 \in \mathbb{T}, \ldots, \sigma_n \in \mathbb{T}$, and an output type $\sigma \in \mathbb{T}$. A *type declaration* for a function symbol f will be written as $f : \sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma$. Type declarations are not types, although they are used for typing purposes. Note, however, that $\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma$ is a type if $f : \sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma$ is a type declaration. We will use the letters f, g, h to denote function symbols.

Given a signature $\mathcal{F}$ and a denumerable set $\mathcal{X}$ of variables, the set of *raw algebraic λ-terms* is defined by $\mathcal{T} := \mathcal{X} \mid (\lambda\mathcal{X} : \mathbb{T}.\mathcal{T}) \mid @(\mathcal{T},\mathcal{T}) \mid \mathcal{F}(\mathcal{T},\ldots,\mathcal{T})$.
Terms of the form $\lambda x : \sigma.u$ are called *abstractions*, while the other terms are said to be *neutral*. For sake of brevity, we will often omit types. $@(u,v)$ denotes the application of u to v. The application operator is allowed to have a variable arity. We call a *partial left-flattening* of the term $@(@(\ldots @(t_1,t_2)\ldots,t_{n-1}),t_n)$, any term of the form $@(@(\ldots @(t_1,t_2)\ldots,t_i),t_{i+1},\ldots,t_n)$. As a matter of convenience, we may write $@(u,v_1,\ldots,v_n)$ for $@(@(\ldots @(u,v_1)\ldots),v_n)$, assuming $n \geq 1$.

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. By $|t|$ we denote the size of t, and by $|t|_v$, the size of t without counting the variables. 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 $s_1,\ldots,s_n$.

2.2 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:
$$\frac{f : \sigma_1 \times \ldots \times \sigma_n \to \sigma \in \mathcal{F} \qquad \Gamma \vdash t_1 : \sigma_1' \equiv \sigma_1 \;\ldots\; \Gamma \vdash t_n : \sigma_n' \equiv \sigma_n}{\Gamma \vdash f(t_1,\ldots,t_n) : \sigma}$$

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

Application:
$$\frac{\Gamma \vdash s : \sigma \to \tau \quad \Gamma \vdash t : \sigma' \equiv \sigma}{\Gamma \vdash @(s,t) : \tau}$$

A term M has type σ in the environment Γ if $\Gamma \vdash M : \sigma$ is provable in the above inference system. A term M is typable in the environment Γ if there exists a unique type σ such that M has type σ in the environment Γ. A term M is typable if it is typable in some environment Γ. Note again that function symbols are uncurried, hence must come along with all their arguments.

The reason to use $\equiv$ is that having a larger set of typable terms allow us to increase the power of the ordering we will define (see the end of the proof of example 3).

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. Substitutions behave as endomorphisms defined on free variables (avoiding captures).

2.3 Higher-order rewrite rules

The rewrite relation which is considered in this paper is the union of the one induced by a set of higher-order rewrite rules and the β-reduction relation both working modulo α-conversion.

We use $\longrightarrow_\beta$ for the β-reduction rule: $@(\lambda x.v, u) \longrightarrow_\beta v\{x \mapsto u\}$. The simply typed λ-calculus is confluent and terminating with respect to β-reductions.

As said, for simplicity reasons, in this work we do not consider η-reductions, although all results can be extended to include them.

A higher-order *term rewrite system* is a set of rewrite rules $R = \{\Gamma \vdash l_i \to r_i\}_i$, 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. Note that usually the terms l_i and r_i will be typable in the system without using the type equivalence $\equiv$, i.e. they will be typable by the system replacing $\equiv$ by syntactic equality $=$ on types.

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 $t = s[r\gamma]_p$ (modulo α-conversion). We denote by $\xrightarrow[R]{*}$ the reflexive, transitive closure of the rewrite relation $\xrightarrow[R]{}$. 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.

2.4 Orderings and quasi-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 and quasi-orderings (see e.g. [DJ90]). For our purpose, a *(strict) ordering*, always denoted by $>$ or $\succ$ (possibly with subscripts), is an irreflexive and transitive relation. An ordering $>$ is *monotonic* if $s > t$ implies $f(\ldots s \ldots) > f(\ldots t \ldots)$; it is *stable* if $s > t$ implies $s\gamma > t\gamma$ for all substitutions γ; and it is *well-founded* if there are no infinite sequences $t_1 > t_2 > \ldots$ An ordering is said to be higher-order when it operates on higher-order terms and is α-compatible: it does not distinguish between α-convertible terms.

A *quasi-ordering*, always denoted by $\succeq$ or $\geq$, is a transitive and reflexive binary relation. Its inverse is denoted by $\preceq$. Its *strict part* $\succ$ is the strict ordering $\succeq \setminus \preceq$ (i.e, $s \succ t$ iff $s \succeq t$ and $s \not\preceq t$). Its *equivalence* $\sim$ is $\succeq \cap \preceq$. Note that $\succeq$ is the disjoint union of $\succ$ and $\sim$.

A quasi-ordering $\succeq$ is *well-founded* if $\succ$ is. It is *stable* if $\succ$ is and $s\sigma \succeq t\sigma$ whenever $s \succeq t$. $\succeq$ is *quasi-monotonic* if $f(\ldots, s, \ldots) \succeq f(\ldots, t, \ldots)$ whenever $s \succeq t$. A quasi-ordering is said to be higher-order when it operates on higher-order terms and its equivalence includes α-conversion. Note that if $\succ$ is a higher-order ordering $\succ \cup =_\alpha$ is a higher-order quasi-ordering whose strict part is $\succ$.

Assume $>_1, \ldots, >_n$ are orderings on sets $S_1, \ldots, S_n$. Then its *lexicographic combination* $(>_1, \ldots, >_n)_{lex}$ is an ordering on $S_1 \times \ldots \times S_n$. We write $>_{lex}$ if all sets and orderings are equal. Similarly the lexicographic combination of quasi-orderings $\succeq_1, \ldots, \succeq_n$ is denoted by $(\succeq_1, \ldots, \succeq_n)_{lex}$ and defined as $s(\succeq_1, \ldots, \succeq_n)_{lex} t$ iff either $s \succ_i t$ for some i and $s \succeq_j t$ for all $j < i$, or $s \succeq_i t$ for all i. If all $>_i$, respectively $\succeq_i$, are well-founded then its lexicographic combination also is. Same happens for stability and α-compatibility.

Assume $\succ$ is an ordering on a set S. Then its *multiset extension*, denoted by $\succ\!\succ$, is an ordering on the set of multisets of elements of S, defined as the transitive closure of: $M \cup \{s\} \succ\!\succ M \cup \{t_1, \ldots, t_n\}$ if $s \succ t_i \ \forall i \in [1..n]$ (using $\cup$ for multiset union). If $\succ$ is well-founded, stable and α-compatible then $\succ\!\succ$ also is.

Definition 1. *A higher-order reduction ordering is a well-founded, monotonic and stable higher-order ordering $>$, such that $\longrightarrow_\beta \ \subseteq \ >$.*

A higher-order quasi-rewrite ordering is a quasi-monotonic and stable higher-order quasi-ordering $\succeq$, such that $\longrightarrow_\beta \ \subseteq \ \succeq$. A higher-order quasi-reduction ordering is in addition well-founded.

Reduction orderings allow us to show that the relation $\longrightarrow_R \cup \longrightarrow_\beta$ is terminating by simply comparing the left-hand and right-hand sides of each rule in R:

Theorem 1. *Let $R = \{\Gamma \ \vdash \ l_i \to r_i\}_{i \in I}$ be a higher-order rewrite system such that $l_i > r_i$ for every $i \in I$. Then the relation $\longrightarrow_R \cup \longrightarrow_\beta$ is strongly normalizing.*

2.5 The Higher-Order Recursive Path Ordering (HORPO)

We present here a restricted version of HORPO (with no status for function symbols) which is enough for our purposes. The HORPO is based on a quasi-ordering $\geq_{\mathcal{F}}$, called *precedence*, on the set of function symbols $\mathcal{F}$ whose strict part $>_{\mathcal{F}}$ is well-founded and whose equivalence is denoted by $=_{\mathcal{F}}$. HORPO compares terms of equivalent type by using the head symbol wrt. the precedence and/or the arguments recursively following a similar structure as RPO in the first-order case.

Additionally, in HORPO, in order to make the ordering more powerful, when comparing two terms s and t, when s is headed by an algebraic function symbol, we cannot only use the arguments of s but also any term in the so called *computable closure* of s, which mainly allows us to introduce abstractions in some cases. The intuition for doing so comes from the strong normalization proof. In that proof, it is crucial to show the computability (in the sense of Tait and Girard's strong normalization proof technique) of the right-hand side term t by using the left-hand side arguments of s, which are assumed to be computable, and the head function symbol. Therefore, instead of using directly the arguments of s, we can use any term obtained applying computability preserving operations to its arguments.

To ease the reading, here we give a subset of the possible computability preserving operations given in [JR99] which is enough for our purposes and gives the flavor of the method. In the following definitions $=$ is considered as α-equality.

Definition 2. *Given a term $s = f(s_1, \ldots, s_n)$, we define its computable closure $\mathcal{CC}(s)$ as $\mathcal{CC}(s, \emptyset)$, where $\mathcal{CC}(s, \mathcal{V})$ with $\mathcal{V} \cap \mathcal{V}ar(s) = \emptyset$, is the smallest set of well-typed terms containing the arguments $s_1, \ldots, s_n$, all variables in $\mathcal{V}$ and closed under the following two operations:*

1. precedence: $h(\overline{u}) \in \mathcal{CC}(s, \mathcal{V})$ *if* $f >_{\mathcal{F}} h$ *and* $\overline{u} \in \mathcal{CC}(s, \mathcal{V})$.
2. abstraction: $\lambda x.u \in \mathcal{CC}(s, \mathcal{V})$ *if* $x \notin \mathcal{V}ar(s) \cup \mathcal{V}$ *and* $u \in \mathcal{CC}(s, \mathcal{V} \cup \{x\})$.

Now we give the definition of HORPO, denoted by $\succ_{horpo}$, adapted from [JR99], where $\succeq_{horpo}$ means the union of $\succ_{horpo}$ and α-conversion.

$$s : \sigma \underset{horpo}{\succ} t : \tau \text{ iff } \sigma \equiv \tau \text{ and}$$

1. $s = f(\overline{s})$ with $f \in \mathcal{F}$, and $u \succeq_{horpo} t$ for some $u \in \overline{s}$
2. $s = f(\overline{s})$ with $f \in \mathcal{F}$ and $t = g(\overline{t})$ with $f >_{\mathcal{F}} g$, and for all $t_i \in \overline{t}$ either $s \succ_{horpo} t_i$ or $u \succeq_{horpo} t_i$ for some $u \in \mathcal{CC}(s)$
3. $s = f(\overline{s})$ and $t = g(\overline{t})$, $f =_{\mathcal{F}} g \in \mathcal{F}$ and $\{\overline{s}\} \succ\!\!\succ_{horpo} \{\overline{t}\}$
4. $s = f(\overline{s})$ with $f \in \mathcal{F}$, $t = @(\overline{t})$ is some partial left-flattening of t, and for all $t_i \in \overline{t}$ either $s \succ_{horpo} t_i$ or $u \succeq_{horpo} t_i$ for some $u \in \mathcal{CC}(s)$
5. $s = @(s_1, s_2)$, $t = @(t_1, t_2)$ and $\{s_1, s_2\} \succ\!\!\succ_{horpo} \{t_1, t_2\}$
6. $s = \lambda x.u$ and $t = \lambda x.v$, and $u \succ_{horpo} v$
7. $s = @(\lambda x.u, v)$ and $u\{x \mapsto v\} \succeq_{horpo} t$

The definition of HORPO is used as a recursive algorithm to check whether a term s is greater than a term t. In case 6 we apply an α-conversion if necessary. The following theorem states that HORPO is a higher-order reduction ordering, which means that if we show that the left hand side of each rule in a rewriting system is greater than the right hand side then we can conclude that the system is terminating for higher-order rewriting.

Theorem 2. *The transitive closure of* $\succ_{horpo}$ *is a higher-order reduction ordering.*

3 A motivating example

In this section we present an example which on the one hand will help us to show how HORPO is used, and on the other hand will exhibit its weakness, since the example cannot be proved terminating although it is, and the need to improve in the direction of semantic orderings.

The example defines the *prefix sum* of a list of natural numbers, i.e. the list with the sum of all prefixes of the given list, using the *m*ap function. We do not include the rules defining the + symbol.

Example 1 *Prefix sum. Let* $V_{\mathbb{T}} = \{Nat, List\}$,
$\mathcal{X} = \{\ x : Nat, xs : List, F : Nat \to Nat\ \}$ *and*
$\mathcal{F} = \{\ [] : List,\ cons : Nat \times List \to List,\ + : Nat \times Nat \to Nat,$
$\quad map : (Nat \to Nat) \times List \to List,\ ps : List \to List\ \}.$

$$map(F, []) \to [] \tag{1}$$
$$map(F, cons(x, xs)) \to cons(@(F, x), map(F, xs)) \tag{2}$$

$$ps([]) \to [] \tag{3}$$
$$ps(cons(x, xs)) \to cons(x, ps(map(\lambda y.x + y, xs))) \tag{4}$$

Now we try to prove that this system is included in HORPO. Rules 1 and 3 hold applying case 1 of the definition of HORPO. For the second rule we need to show that $map(F, cons(x, xs)) : List \succ_{horpo} cons(@(F, x), map(F, xs)) : List$. In this case we can see that the only possibility is to have $map >_{\mathcal{F}} cons$ and apply case 2. Then we check recursively that (1) $map(F, cons(x, xs)) : List \succ_{horpo} @(F, x) : Nat$ and (2) $map(F, cons(x, xs)) : List \succ_{horpo} map(F, xs) : List$. For (1) we apply case 4, since $F \in \mathcal{CC}(map(F, cons(x, xs)))$ and $map(F, cons(x, xs)) : List \succ_{horpo} x : Nat$, applying twice case 1. Finally, to prove (2) we apply case 3 which holds since $cons(x, xs) : List \succ_{horpo} xs : List$ by case 1.

For the last rule, we can only apply case 2 taking $ps >_{\mathcal{F}} cons$. This requires that $ps(cons(x, xs)) : List \succ_{horpo} x : Nat$, which holds applying twice case 1, and also $ps(cons(x, xs)) : List \succ_{horpo} ps(map(\lambda y.x + y, xs)) : List$. For the latter we apply case 3, which requires $cons(x, xs) : List \succ_{horpo} map(\lambda y.x + y, xs) : List$. To prove this we need $cons >_{\mathcal{F}} map$ and apply case 2, showing that $\lambda y.x+y \in \mathcal{CC}(cons(x, xs))$ taking $cons >_{\mathcal{F}} +$, and $cons(x, xs) : List \succ_{horpo} xs : List$ using case 1.

Unfortunately, to show that the second rule is in HORPO we need to take $map >_{\mathcal{F}} cons$ and to show the last rule we need $cons >_{\mathcal{F}} map$, which of course cannot be the case if $>_{\mathcal{F}}$ is well-founded. Note that the considered system cannot be proved either using the definition of HORPO in [JR99], hence the problem is not due to the simplified version of HORPO we are considering here.

Now, if we look at the example, the intuition behind its termination comes from the fact that in all recursive calls the size of the list parameter decrease. This somehow means that this parameter should play an important role when comparing the terms, since the use of the head symbol is not enough. Generalizing path orderings to using more information about the term than only the head symbol is done by means of *semantic path orderings* [KL80].

4 The Higher-Order Semantic Path Ordering (HOSPO)

We present now HOSPO, which generalizes HORPO by using a well-founded stable higher-order quasi-ordering $\succeq_Q$, which does not need to include β-reduction, instead of the precedence $\geq_{\mathcal{F}}$. We also adapt in the same way the computable closure. Note that, although we can fully adapt the definition of the computable closure given in [JR99], for simplicity reasons, we will provide only an adapted version of the restricted computable closure given in section 2.5.

Definition 3. *Given a term $s = f(s_1, \ldots, s_n)$, we define its computable closure $\mathcal{CC}(s)$ as $\mathcal{CC}(s, \emptyset)$, where $\mathcal{CC}(s, \mathcal{V})$ with $\mathcal{V} \cap \mathcal{V}ar(s) = \emptyset$, is the smallest set of well-typed terms containing all variables in $\mathcal{V}$ and all terms in $\{s_1, \ldots, s_n\}$, and closed under the following operations:*

1. *quasi-ordering: $h(\overline{u}) \in \mathcal{CC}(s, \mathcal{V})$ if $f(\overline{s}) \succ_Q h(\overline{u})$ and $\overline{u} \in \mathcal{CC}(s, \mathcal{V})$.*
2. *abstraction: $\lambda x.u \in \mathcal{CC}(s, \mathcal{V})$ if $x \notin \mathcal{V}ar(s) \cup \mathcal{V}$ and $u \in \mathcal{CC}(s, \mathcal{V} \cup \{x\})$.*

Now we give the definition of HOSPO, where $\succeq_{hospo}$ means the union of $\succ_{hospo}$ and α-conversion.

Definition 4. $s : \sigma \succ_{hospo} t : \tau$ *iff* $\sigma \equiv \tau$ *and*

1. $s = f(\overline{s})$ *with* $f \in \mathcal{F}$, $u \succeq_{hospo} t$ *for some* $u \in \overline{s}$.
2. $s = f(\overline{s})$ *and* $t = g(\overline{t})$ *with* $f, g \in \mathcal{F}$, $s \succ_Q t$, *and* *for all* $t_i \in \overline{t}$ *either* $s \succ_{hospo} t_i$ *or* $u \succeq_{hospo} t_i$ *for some* $u \in \mathcal{CC}(s)$.
3. $s = f(\overline{s})$ *and* $t = g(\overline{t})$ *with* $f, g \in \mathcal{F}$, $s \succeq_Q t$ *and* $\{\overline{s}\} \succ\!\!\succ_{hospo} \{\overline{t}\}$
4. $s = f(\overline{s})$, $f \in \mathcal{F}$, $@(\overline{t})$ *is some partial left-flattening of* t, *and* *for all* $t_i \in \overline{t}$ *either* $s \succ_{hospo} t_i$ *or* $u \succeq_{hospo} t_i$ *for some* $u \in \mathcal{CC}(s)$.
5. $s = @(s_1, s_2)$, $t = @(t_1, t_2)$, $\{s_1, s_2\} \succ\!\!\succ_{hospo} \{t_1, t_2\}$
6. $s = \lambda x.u$, $t = \lambda x.v$, $u \succ_{hospo} v$
7. $s = @(\lambda x.u, v)$ *and* $u\{x \mapsto v\} \succeq_{hospo} t$

In case 6 we apply an α-conversion if necessary. Note that $\succeq_Q$ is only used in cases 2 and 3, where the head symbol of both terms is in $\mathcal{F}$. On the other hand, case 7 captures β-reduction at top position, and since, in general, HOSPO is not monotonic, it may be the case that HOSPO does not include β-reduction at any position.

The resulting ordering is shown to be well-defined by comparing pairs of terms $\langle t, s\rangle$ in the well-founded ordering $(\rhd, \longrightarrow_\beta \cup \rhd)_{lex}$.

Lemma 1. $\succ_{hospo}$ *is stable under substitutions.*

Proof. First we show that $u \in \mathcal{CC}(s)$ implies that $u\gamma \in \mathcal{CC}(s\gamma)$ for every substitution γ.

We prove that if $u \in \mathcal{CC}(s, \mathcal{V})$ with $\mathcal{V} \cap (\mathcal{V}ar(s) \cup \mathcal{V}ar(s\gamma) \cup Dom(\gamma)) = \emptyset$, then $u\gamma \in \mathcal{CC}(s\gamma, \mathcal{V})$, by induction on the definition of $\mathcal{CC}(s, \mathcal{V})$ and distinguishing cases according to the definition of the computational closure.

- If $u \in \mathcal{V}$ then $u\gamma = u \in \mathcal{V}$. If u is an argument of s then $u\gamma$ is an argument of s.
- Let $u \in \mathcal{CC}(s, \mathcal{V})$ by case 1, that is, $u = h(\overline{u})$ such that $f(\overline{s}) \succ_Q h(\overline{u})$ and $\overline{u} \in \mathcal{CC}(s, \mathcal{V})$. By induction hypothesis $\overline{u}\gamma \in \mathcal{CC}(s\gamma, \mathcal{V})$, and by stability under substitutions of $\succ_Q$, $f(\overline{s})\gamma \succ_Q h(\overline{u})\gamma = h(\overline{u}\gamma)$. Therefore, $h(\overline{u})\gamma \in \mathcal{CC}(s\gamma, \mathcal{V})$ by case 1.
- Let $u = \lambda x.v \in \mathcal{CC}(s, \mathcal{V})$ by case 2, that is, $x \notin \mathcal{V}ar(s) \cup \mathcal{V}$ and $v \in \mathcal{CC}(s, \mathcal{V} \cup \{x\})$. To the price of renaming x if necessary, we can assume in addition that $x \notin \mathcal{V}ar(s\gamma) \cup Dom(\gamma)$, and therefore, $x \in \mathcal{X} \setminus (\mathcal{V} \cup \mathcal{V}ar(s) \cup \mathcal{V}ar(s\gamma) \cup Dom(\gamma))$. By induction hypothesis, $v\gamma \in \mathcal{CC}(s\gamma, \mathcal{V} \cup \{x\})$, and since $x \notin \mathcal{V}ar(s\gamma) \cup \mathcal{V}$ then $\lambda x.v\gamma \in \mathcal{CC}(s\gamma, \mathcal{V})$. Using now the hypothesis that $x \notin Dom(\gamma)$, $u\gamma = \lambda x.v\gamma$, and we are done.

Now we can prove stability under substitutions of $\succ_{hospo}$, that is, $s : \sigma \succ_{hospo} t : \tau$ implies $s\gamma \succ_{hospo} t\gamma$ for all substitution γ. The proof is done by induction on the pair $\langle t, s\rangle$ ordered lexicographically by $(\rhd, \longrightarrow_\beta \cup \rhd)_{lex}$ and distinguishing cases according to the definition of $\succ_{hospo}$.

- $s \succ_{hospo} t$ by case 1, that is, $s = f(\overline{s})$ with $f \in \mathcal{F}$, $s_i \succeq_{hospo} t$ for some $s_i \in \overline{s}$. By induction hypothesis, $s_i\gamma \succeq_{hospo} t\gamma$, and hence, $s\gamma \succ_{hospo} t\gamma$ by case 1.

- $s \succ_{hospo} t$ by case 2, that is, $s = f(\overline{s})$ and $t = g(\overline{t})$ with $f, g \in \mathcal{F}$, $s \succ_Q t$ and $\forall t_i \in \overline{t}\ \ s \succ_{hospo} t_i\ \ or\ \ u \succeq_{hospo} t_i$ for some $u \in \mathcal{CC}(s)$.
By stability of $\succ_Q$, $s\gamma \succ_Q t\gamma$, and by induction hypothesis, if $s \succ_{hospo} t_i$ then $s\gamma \succ_{hospo} t_i\gamma$, and if $u \succeq_{hospo} t_i$ then $u\gamma \succeq_{hospo} t_i\gamma$ and since above we have proved that $u \in \mathcal{CC}(s)$ implies $u\gamma \in \mathcal{CC}(s\gamma)$ then we can conclude $s\gamma \succ_{hospo} t\gamma$ by case 2.
- $s \succ_{hospo} t$ by case 3, that is, $s = f(\overline{s})$ and $t = g(\overline{t})$ with $f, g \in \mathcal{F}$, $s \succeq_Q t$ and $\{\overline{s}\} \succ\!\!\succ_{hospo} \{\overline{t}\}$. By stability of $\succeq_Q$, $s\gamma \succeq_Q t\gamma$, and by induction hypothesis, $\{\overline{s\gamma}\} \succ\!\!\succ_{hospo} \{\overline{t\gamma}\}$. Therefore, $s\gamma \succ_{hospo} t\gamma$ by case 3.
- $s \succ_{hospo} t$ by case 4, that is, $s = f(\overline{s})$, $f \in \mathcal{F}$, $@(\overline{t})$, $s \succ_Q t$ and $\forall t_i \in \overline{t}\ \ s \succ_{hospo} t_i\ \ or\ \ u \succeq_{hospo} t_i$ for some $u \in \mathcal{CC}(s)$.
If $s \succ_{hospo} t_i$ then $s\gamma \succ_{hospo} t_i\gamma$, and if $u \succeq_{hospo} t_i$ then $u\gamma \succeq_{hospo} t_i\gamma$ and, by the above property, $u\gamma \in \mathcal{CC}(s\gamma)$ since $u \in \mathcal{CC}(s)$. If $@(\overline{t})$ is a partial left-flattening of t then $@(\overline{t\gamma})$ is a partial left-flattening of $t\gamma$. Therefore, $s\gamma \succ_{hospo} t\gamma$ by case 4.
- $s \succ_{hospo} t$ by case 5, that is, $s = @(s_1, s_2)$, $t = @(t_1, t_2)$, $\{s_1, s_2\} \succ\!\!\succ_{hospo} \{t_1, t_2\}$. By induction hypothesis, $\{s_1\gamma, s_2\gamma\} \succ\!\!\succ_{hospo} \{t_1\gamma, t_2\gamma\}$, and therefore, $s\gamma \succ_{hospo} t\gamma$ by case 5.
- $s \succ_{hospo} t$ by case 6, that is, $s = \lambda x.u$, $t = \lambda x.v$, $u \succ_{hospo} v$. By induction hypothesis, $u\gamma \succ_{hospo} v\gamma$. Therefore, $s\gamma \succ_{hospo} t\gamma$ by case 6.
- $s \succ_{hospo} t$ by case 7, that is, $s = @(\lambda x.u, v)$, $u\{x \mapsto v\} \succeq_{hospo} t$. By stability of the β-reduction, $s\gamma = @(\lambda x.u\gamma, v\gamma) \to_\beta u\gamma\{x \mapsto v\gamma\}$ and, by induction hypothesis, $u\gamma\{x \mapsto v\gamma\} \succeq_{hospo} t\gamma$. Therefore, $s\gamma \succ_{hospo} t\gamma$ by case 7.

Lemma 2. *$\succ_{hospo}$ is α-compatible.*

Proof. First we prove that if $u \in \mathcal{CC}(s, \mathcal{V})$ with $\mathcal{V} \cap \mathcal{V}ar(s) = \emptyset$ then there is a some set of variables $\mathcal{V}'$, a one to one substitution γ from $\mathcal{V}$ to $\mathcal{V}'$ and a term u', s.t. $u\gamma =_\alpha u' \in \mathcal{CC}(s', \mathcal{V})$. In particular, for $\mathcal{V} = \emptyset$, we have if $u \in \mathcal{CC}(s)$ then $u =_\alpha u' \in \mathcal{CC}(s')$.

We proceed by induction on the definition of the computable closure. If u is x in $\mathcal{V}$ then u' is the image of x in γ. If u is an argument of s then there is some u' argument of u' with $u =_\alpha u'$ (note that in this case u does not contain any variable in $\mathcal{V}$. Otherwise, if $u = h(\overline{u}) \in \mathcal{CC}(s, \mathcal{V})$ by case 1 then $f(\overline{s}) \succ_Q h(\overline{u})$ and $\overline{u} \in \mathcal{CC}(s, \mathcal{V})$. Then, by induction hypothesis there are terms $\overline{u'} \in \mathcal{CC}(s', \mathcal{V}')$ with $\overline{u}\gamma =_\alpha \overline{u'}$ and since $\succ_Q$ is α-compatible and stable under substitutions $f(\overline{s'}) \succ_Q h(\overline{u'})$, which implies $u\gamma =_\alpha h(\overline{u'}) \in \mathcal{CC}(s', \mathcal{V}')$, by case 1. Finally if $u = \lambda x.v \in \mathcal{CC}(s, \mathcal{V})$ by case 2, then $x \notin \mathcal{V}ar(s) \cup \mathcal{V}$ and $v \in \mathcal{CC}(s, \mathcal{V} \cup \{x\})$. Then by induction hypothesis there is some set $\mathcal{V}' \cup \{x'\}$, a substitution γ' of the form $\gamma \cup \{x \mapsto x'\}$ and a term v' s.t. $v\gamma' =_\alpha v' \in \mathcal{CC}(s', \mathcal{V}' \cup \{x'\})$, which implies $u\gamma =_\alpha \lambda x'.v' \in \mathcal{CC}(s', \mathcal{V}')$, by case 2.

Now we prove the α-compatibility of $\succ_{hospo}$, that is, $s' =_\alpha s \succ_{hospo} t =_\alpha t'$ implies $s' \succ_{hospo} t'$. We proceed by induction on the pair $\langle t, s\rangle$ ordered lexicographically by $(\rhd, \to_\beta \cup \rhd)_{lex}$. All cases follow easily by induction, using the property above in cases 2 and 4. We develop case 6, i.e. we have $s = \lambda x.u \succ_{hospo} \lambda x.v = t$ (possibly after applying an α-conversion step on t) and we want to prove $s' \succ_{hospo} t'$. Let $s' = \lambda y.u'$ with $u'\{y \mapsto x\} =_\alpha u$, then (possibly after applying an α-conversion step on t') we have $t' = \lambda y.v'$ with $v'\{y \mapsto x\} =_\alpha v$. By induction hypothesis, we have $u'\{y \mapsto x\} \succ_{hospo} v'\{y \mapsto x\}$, and by stability under substitutions replacing again x by y we have $u' \succ_{hospo} v'$, and hence $s' = \lambda y.u' \succ_{hospo} \lambda y.v' = t'$.

To prove well-foundedness of the ordering we follow the Tait and Girard's computability predicate proof method. We denote by $[\![\sigma]\!]$ the computability predicate of type σ.

Definition 5. *The family of type interpretations $\{[\![\sigma]\!]\}_{\sigma \in \mathcal{T}_S}$ is the family of subsets of the set of typed terms whose elements are the least sets satisfying the following properties:*

1. *If σ is a basic type, then $s : \sigma \in [\![\sigma]\!]$ iff $\forall t : \tau$ s.t. $s \succ_{hospo} t$, $t \in [\![\tau]\!]$.*
2. *If $s : \sigma = \tau \to \rho$ then $s \in [\![\sigma]\!]$ iff $@(s,t) \in [\![\rho]\!]$ for every $t \in [\![\tau']\!]$, with $\tau' \equiv \tau$.*

The above definition is based on a lexicographic combination of an induction on the size of the type and a fixpoint computation for basic types. The existence of a least fixpoint is ensured by the monotonicity of the underlying family of functionals (indexed by the set of basic types) with respect to set inclusion (for each basic type). Note that for basic types this definition can be seen as a closure wrt. case 1, taking as initial set for each basic type the set of minimal, wrt. $\succ_{hospo}$, terms (which includes the variables). A typed term s of type σ is said to be computable if $s \in [\![\sigma]\!]$. A vector $\overline{s}$ of terms is computable iff so are all its components.

Lemma 3. *Let $\sigma = \sigma_1 \to \ldots \to \sigma_n \to \tau$ where $n > 0$ and τ is a data type. Then $s \in [\![\sigma]\!]$ iff $@(s, t_1, \ldots, t_n) \in [\![\tau]\!]$ for all $t_1 \in [\![\sigma'_1]\!], \ldots, t_n \in [\![\sigma'_n]\!]$, with $\sigma_i \equiv \sigma'_i$ for all $i \in \{1 \ldots n\}$.*

Proof. By induction on n and applying case 2 of the definition.

Property 1 *Computability properties.*

1. *Every computable term is strongly normalizing.*
2. *If s is computable and $s \succ_{hospo} t$ then t is computable.*
3. *A neutral term s is computable iff t is computable for every t s.t. $s \succ_{hospo} t$.*
4. *If $\overline{t}$ is a vector of at least two computable terms s.t. $@(\overline{t})$ is a typed term, then $@(\overline{t})$ is computable.*
5. *$\lambda x : \sigma.u$ is computable iff $u\{x \mapsto w\}$ is computable for every computable term $w : \alpha \equiv \sigma$.*

Proof. 1. Let $s \in [\![\sigma]\!]$. The proof that s is strongly normalizing is by induction on the definition of computable terms. Variables are strongly normalizing. If σ is a basic type then, by definition, s computable implies t computable for every t such that $s \succ_{hospo} t$. By induction hypothesis t is strongly normalizing, and hence s is strongly normalizing. If $\sigma = \tau \to \rho$ then, by definition, $@(s,t)$ is computable for every t computable. Since a variable x is computable then $@(s,x)$ is computable and by induction hypothesis is strongly normalizing, and therefore its subterm s is strongly normalizing.

2. Let $s \in [\![\sigma]\!]$ and $s : \sigma \succ_{hospo} t : \tau$. We prove that $t \in [\![\tau]\!]$ by induction on the length (number of arrows) of the type. If σ is a basic type then the property follows from the case 1 of the definition. If $\sigma = \rho_1 \to \rho_2$ then by definition of $\succ_{hospo}$, $\tau = \rho'_1 \to \rho'_2$ with $\rho_1 \equiv \rho'_1$ and $\rho_2 \equiv \rho'_2$. Now for all $u \in [\![\rho]\!]$ with $\rho \equiv \rho_1$ we have $@(s,u)[\![\rho_2]\!]$, and, since $s \succ_{hospo} t$, by case 5, we have $@(s,u) : \rho_2 \succ_{hospo} @(t,u) : \rho'_2$. Hence, by induction hypothesis, $@(t,u) : \rho'_2$ for every $u \in [\![\rho]\!]$, with $\rho \equiv \rho'_1$, which implies, by case 2 of the definition, that $t \in [\![\tau]\!]$.
3. The only if part is property 1.2. For the if part, if $s : \sigma$ with σ a basic type then, the property follows from the definition. Assume now that $\sigma = \sigma_1 \to ... \to \sigma_n \to \tau$ where $n > 0$ and τ is a basic type. By lemma 3, $s \in [\![\sigma]\!]$ iff $@(s, t_1, ..., t_n) \in [\![\tau]\!]$ for all $t_1 \in [\![\sigma'_1]\!]$,...$t_n \in [\![\sigma'_n]\!]$. Since τ is a basic type then $@(s, t_1, ..., t_n) \in [\![\tau]\!]$ iff $v \in [\![\rho]\!]$ for all $v : \rho$ s.t. $@(s, t_1, ..., t_n) \succ_{hospo} v$. Therefore, now we will prove $v \in [\![\rho]\!]$. Since all reducts of s are computable and, by assumption, $t_1, ..., t_n$ are computable, then we use induction on the multiset $\{s, t_1, ..., t_n\}$ ordered by $\succ\!\!\succ_{hospo}$ which is well-founded by property 1.1.
Since s is neutral, $\succ_{hospo}$ by case 7 do not apply. Hence $@(s, t_1, ..., t_n) \succ_{hospo} v$ by case 5, that is, $v = @(v_1, v_2)$ and $\{@(s, t_1...t_{n-1}), t_n\} \succ\!\!\succ_{hospo} \{v_1, v_2\}$. By typing reasons we have $@(s, t_1, ...t_{n-1}) \succeq_{hospo} v_1$, $t_n \succeq_{hospo} v_2$ and at least, one of them is strict. If $@(s, t_1, ...t_{n-1}) \succ_{hospo} v_1$ then, by induction hypothesis, v_1 is computable, and since $v2$ is computable by property 1.2, we have that $v = @(v_1, v_2)$ is computable by lemma 3. Otherwise $@(s, t_1, ...t_{n-1}) = v_1$ and $t_n \succ_{hospo} v_2$, and hence, $v = @(s, t_1, ...t_{n-1}, v_2)$ which is computable by induction hypothesis.
4. Let $\bar{t}$ be $(t_1, ..., t_n)$, $\bar{t}$ computable and $t_1 : \alpha_1 = \alpha_2 \to ... \to \alpha_n \to ...\alpha_m \to \beta$, β basic type and $m \succeq n$. By lemma 3, $@(t_1...t_n...t_m)$ is computable for all $t_{n+1}...t_m$ computable. If $n = m$ then we have $@(\bar{t})$ is computable. Otherwise $m \succ n$ and since $@(t_1...t_m) = @(@(t_1...t_n)...t_m)$ then, by lemma 3, $@(t_1...t_n)$ is computable.
5. First we prove the only if part. By case 2 of the definition of the interpretations, $\lambda x : \sigma.u \in [\![\sigma \to \tau]\!]$ implies $@(\lambda x.u, w) \in [\![\tau]\!]$ for all $w \in [\![\sigma]\!]$. By property 1.2, $@(\lambda x.u, w) \succ_{hospo} v$ implies v is computable. Therefore, $u\{x \mapsto w\}$ is computable since $@(\lambda x.u, w) \succ_{hospo} u\{x \mapsto w\}$ by case 7. For the if part we will prove that $u\{x \mapsto w\} \in [\![\tau]\!]$ for every $w \in [\![\sigma]\!]$ implies $@(\lambda x.u, w) \in [\![\tau]\!]$ for every $w \in [\![\sigma]\!]$ since, by case 2 of the definition of the interpretations, this implies $\lambda x : \sigma.u$ is computable.
We know u is computable since a variable is computable and hence $u\{x \mapsto x\} = u$ is computable by assumption. Therefore, by property 1.1 we have u and w are strongly normalizing and hence we can use induction on the multiset $\{\lambda x.u, w\}$ ordered by $\succ\!\!\succ_{hospo}$.
Since $@(\lambda x.u, w)$ is a neutral term, by property 1.3 it is computable if v is computable for all v s.t. $@(\lambda x.u, w) \succ_{hospo} v$. There are two cases:
   - $@(\lambda x.u, w) \succ_{hospo} @(v', v'')$ by case 5, that is $\{\lambda x.u, w\} \succ\!\!\succ_{hospo} \{v', v''\}$. We distinguish three cases:
     - $v' = \lambda x.u$ and $w \succ_{hospo} v''$. Then v'' is computable by property 1.2 and hence, by induction hypothesis, $@(\lambda x.u, v'')$ is computable.

- If v' and v'' are reducts of w then, since w is computable, its reducts are computable by property 1.2. Therefore, $@(v', v'')$ is computable by lemma 3.
- Otherwise, for typing reasons, $\lambda x.u \succ_{hospo} v' = \lambda x.u'$ with $u \succ_{hospo} u'$ and $w \succeq_{hospo} v''$. Since, by assumption, $u\{x \mapsto w\}$ is computable for all w computable and, by stability of $\succ_{hospo}$, $u\{x \mapsto w\} \succ_{hospo} u'\{x \mapsto w\}$, then $u'\{x \mapsto w\}$ is computable by property 1.2. Hence, $v = @(\lambda x.u', v'')$ is computable by induction hypothesis.

– $@(\lambda x.u, w) \succ_{hospo} v$ by case 7, that is $u\{x \mapsto w\} \succeq_{hospo} v$. Since $u\{x \mapsto w\}$ is computable by assumption, then v is computable by property 1.2.

Note that variables are computable as a consequence of Property 1.3.

The precise assumption of the following property comes from the ordering used in the proof by induction of Lemma 4 and gives some intuition about the definition of the computable closure.

Property 2 *Assume $\overline{s} : \overline{\tau}$ is computable, as well as every term $h(\overline{u})$ with $\overline{u}$ computable and $f(\overline{s}) \succ_Q h(\overline{u})$. Then every term in $\mathcal{CC}(f(\overline{s}))$ is computable.*

Proof. Let s be $f(\overline{s})$. We prove $u\gamma$ is computable for every computable substitution γ of domain $\mathcal{V}$ and every $u \in \mathcal{CC}(s, \mathcal{V})$ such that $\mathcal{V} \cap \mathcal{V}ar(s) = \emptyset$. We obtain the result by taking $\mathcal{V} = \emptyset$. We proceed by induction on the definition of $\mathcal{CC}(s, \mathcal{V})$.

For the basic case: if $u \in \overline{s}$, we conclude by assumption that $\overline{s}$ 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.

For the induction step, we distinguish cases according to the definition of the computable closure:

1. case 1: $u = h(\overline{u})$, $\overline{u} \in \mathcal{CC}(s, \mathcal{V})$ and $f(\overline{s}) \succ_Q h(\overline{u})$. By induction hypothesis, $\overline{u}\gamma$ is computable and since, by stability of $\succ_Q$, $f(\overline{s})\gamma \succ_Q h(\overline{u})\gamma$, then $u\gamma$ is computable by assumption.
2. case 2: let $u = \lambda x.v$ with $x \notin \mathcal{V} \cup \mathcal{V}ar(s)$ and $v \in \mathcal{CC}(s, \mathcal{V} \cup \{x\})$. To the price of possibly renaming x, we can assume without loss of generality that $x \notin Dom(\gamma)$. Therefore $\mathcal{V} \cup \{x\} \cap \mathcal{V}ar(s) = \emptyset$, and given an arbitrary computable term w, $\gamma' = \gamma \cup \{x \mapsto w\}$ is a computable substitution of domain $\mathcal{V} \cup \{x\}$. By induction hypothesis, $v\gamma'$ is therefore computable, and by property 1.5 $(\lambda x.v)\gamma$ is computable.

Lemma 4. *Let $f : \overline{\sigma} \to \tau \ \in \mathcal{F}$ and $\overline{t} : \overline{\tau} \equiv \overline{\sigma}$ be a set of terms. If $\overline{t}$ is computable, then $f(\overline{t})$ is computable.*

Proof. The proof is done by induction on the ordering $\langle \succ_Q, \succ\!\!\succ_{hospo} \rangle$ operating on pairs $\langle f(\overline{t}), \overline{t} \rangle$. This ordering is well-founded since we are assuming that $\overline{t}$ is computable and hence strongly normalizing. Note that in the assumption of Property 2 we are only using the first component of the induction ordering. By using both components, we can improve the computable closure, as done in [JR99], adding new cases (see section 7).

Since $f(\overline{t})$ is neutral, by property 1.3, it is computable iff every s such that $f(\overline{t}) \succ_{hospo} s$ is computable which we prove by an inner induction on $|s|$. We distinguish several cases according to the definition of $\succ_{hospo}$.

- $f(\overline{t}) \succ_{hospo} s$ by case 1, that is, $t_i \succeq_{hospo} s$ for some $t_i \in \overline{t}$. Since t_i is computable, then s is computable by property 1.2.
- $t = f(\overline{t}) \succ_{hospo} g(\overline{s}) = s$ by case 2, that is, $f, g \in \mathcal{F}$, $f(\overline{t}) \succ_Q g(\overline{s})$ and $\forall s_i \in \overline{s}\ \ t \succ_{hospo} s_i \ \ or \ \ u \succeq_{hospo} s_i$ for some $u \in \mathcal{CC}(t)$. If $t \succ_{hospo} s_i$, by the inner induction s_i is computable. Otherwise $u \succeq_{hospo} s_i$. Since $\overline{t}$ is computable and for every computable $\overline{s}$ and $f(\overline{t}) \succ_Q g(\overline{s})$, by the outer induction hypothesis, $g(\overline{s})$ is computable, then by property 2 every term in $\mathcal{CC}(t)$ is computable. Hence u is computable and, by property 1.2, s_i is computable.
  Therefore $\overline{s}$ is computable, and since $f(\overline{t}) \succ_Q g(\overline{s})$ then, by the induction hypothesis, $g(\overline{s})$ is computable.
- $t = f(\overline{t}) \succ_{hospo} g(\overline{s}) = s$ by case 3, that is, $f, g \in \mathcal{F}$, $t \succeq_Q s$ and $\{\overline{t}\} \succ\!\!\succ_{hospo} \{\overline{s}\}$. By definition of the multiset comparison, for every $s_i \in \overline{s}$ there is some $t_j \in \overline{t}$ s.t. $t_j \succeq_{hospo} s_i$, and since $\overline{t}$ is computable then, by property 1.2, s_i is computable. Therefore, s is computable by the outer induction hypothesis.
- $t = f(\overline{t}) \succ_{hospo} s$ by case 4, that is, $f \in \mathcal{F}$, $@(\overline{s})$, $f(\overline{t}) \succ_Q s$ and $\forall s_i \in \overline{s}\ \ t \succ_{hospo} s_i \ \ or \ \ u \succeq_{hospo} s_i$ for some $u \in \mathcal{CC}(t)$. We conclude $\overline{s}$ is computable by the same arguments used in the second case.

Lemma 5. *$\succ_{hospo}$ is well-founded.*

Proof. First we prove that $t\gamma$ is computable for every typed term t and computable substitution γ. The proof is by induction on $|t|_v$.

- If t is a variable x then $x\gamma$ is computable by assumption.
- If $t = \lambda x.u$ then, by property 1.5, $t\gamma$ is computable if $u\gamma\{x \mapsto w\}$ is computable for every well-typed computable term w. Let $\delta = \gamma \cup \{x \mapsto w\}$ then we have $u\gamma\{x \mapsto w\} = u(\gamma \cup \{x \mapsto w\}) = u\delta$ since x may not occur in γ. Since δ is computable, and $|t|_v > |u|_v$, by induction hypothesis, $u\delta$ is computable and hence $t\gamma$ is computable.
- $t = @(t_1, t_2)$ or $t = f(t_1...t_n)$ and some t_i is not a variable. Let $s = @(x_1, x_2)$ or $s = f(x_1...x_n)$ and $\delta = \{x_i \mapsto t_i\gamma | i = 1..n\}$. By induction hypothesis $t_i\gamma$ is computable for all i, hence δ is computable. Since not all t_i are variables then $|t|_v > |s|_v$ and, by induction hypothesis $s\delta$ is computable and since $s\delta = t\gamma$ then $t\gamma$ is computable.
- $t = @(x_1, x_2)$ then $t\gamma$ is computable by property 1.4.
- $t = f(x_1, ..., x_n)$ then $t\gamma$ is computable by lemma 4.

Note that for the empty substitution γ we have that all typed terms are computable and hence, by property 1.4, strongly normalizing.

5 A monotonic Higher Order semantic path ordering.

In this section we present a monotonic version of HOSPO, called MHOSPO, and show how it can be used in practice.

To define MHOSPO we need an additional quasi-ordering $\succeq_I$ as ingredient. This quasi-ordering is used to ensure monotonicity, and due to this we need to require some properties on it.

Definition 6. *We say that* $\succeq_I$ *is* quasi-monotonic on $\succeq_Q$ *(or* $\succeq_Q$ *is* quasi-monotonic wrt. $\succeq_I$*) if*

$$s \succeq_I t \quad \textit{implies} \quad f(\ldots s \ldots) \succeq_Q f(\ldots t \ldots)$$

for all terms s *and* t *and function symbols* f*.*

A pair $\langle \succeq_I, \succeq_Q \rangle$ *is called a* higher-order quasi-reduction pair *if* $\succeq_I$ *is a higher-order quasi-rewrite ordering,* $\succeq_Q$ *is a well-founded stable higher-order quasi-ordering, and* $\succeq_I$ *is quasi-monotonic on* $\succeq_Q$*.*

Now we give the definition of MHOSPO.

Definition 7. *Let* $\langle \succeq_I, \succeq_Q \rangle$ *be a higher-order quasi-reduction pair.*

$$s \succ_{mhospo} t \textit{ iff } s \succeq_I t \textit{ and } s \succ_{hospo} t.$$

Theorem 3. *The transitive closure of* $\succ_{mhospo}$ *is a higher-order reduction ordering.*

Well-foundedness follows from the fact that $\succ_{mhospo} \subseteq \succ_{hospo}$ and $\succ_{hospo}$ is well-founded. Stability and α-compatibility follow respectively from the stability and α-compatibility of $\succeq_I$ and $\succ_{hospo}$. Monotonicity follows directly from the fact that $\succeq_I$ is quasi-monotonic on $\succeq_Q$ and includes β-reduction, and cases 3, 5 and 6 of $\succ_{hospo}$. Finally, to prove that $\longrightarrow_\beta \subseteq \succ_{mhospo}$, we use monotonicity, case 7 of HOSPO and the fact that $\succeq_I$ includes β-reduction.

Note that HORPO is a particular case of MHOSPO, which is obtained by taking $\succeq_I$ as $s \succeq_I t$ for all s and t, which has an empty strict part (and trivially fulfils all required properties), and $\succeq_Q$ as a precedence.

In order to make MHOSPO useful in practice we need general methods to obtain quasi-reduction pairs $\langle \succeq_I, \succeq_Q \rangle$. We will first provide possible candidates for the quasi-ordering $\succeq_I$ and then show how to obtain a $\succeq_Q$ forming a pair.

5.1 Building $\succeq_I$

We consider $\succeq_I$ obtained by combining an interpretation I on terms with some higher-order quasi-reduction ordering $\succeq_B$, which is called the *basic* quasi-ordering, i.e. $s : \sigma \succeq_I t : \tau$ if and only if $I(s : \sigma) \succeq_B I(t : \tau)$ (note that hence we also have that $s : \sigma \succ_I t : \tau$ if and only if $I(s : \sigma) \succ_B I(t : \tau)$). An obvious candidate for $\succeq_B$ is (the transitive closure of) HORPO union α-conversion. For the interpretation I, as a general property, we require the preservation of the typability of terms and the quasi-monotonicity, stability, α-compatibility and the inclusion of β-reduction of the basic quasi-ordering $\succeq_B$. Note that, since $\succeq_B$ is well-founded, the obtained $\succeq_I$ is also well-founded and hence it is a quasi-reduction ordering, although $\succeq_I$ is only required to be a quasi-rewrite ordering. The reason for adding the well-foundedness property to $\succeq_I$ in this construction will become clear later when building the quasi-reduction pairs $\langle \succeq_I, \succeq_Q \rangle$.

Below some suitable such interpretations obtained by adapting usual interpretations for the first-order case are provided. We consider interpretations that are mappings from typed terms to typed terms. As a matter of simplicity we have considered only interpretations to terms in the same given signature and set of types. Note that we can enlarge our signature (or the set of types) if new symbols (or types) are needed in the interpretation. On the other hand if we consider interpretations to terms in a new signature and set of types then we not only need to interpret the terms but also the types.

Each symbol $f : \sigma_1 \times ... \times \sigma_n \to \sigma$ can be interpreted either by (1) a projection on a single argument of an equivalent type to the output type of f, denoted by the pair $(f(x_1, \ldots, x_n), x_i)$ with $x_i : \tau$ and $\sigma \equiv \tau$, or else by (2) a function symbol f_I, with an equivalent output type to f, applied to a sequence obtained from the arguments of f preserving the typability of the term, denoted by the pair $(f(x_1, \ldots, x_n), f_I(x_{i_1}, \ldots, x_{i_k}))$, for some $k \geq 0$, $i_1, \ldots, i_k \in \{1, ..., n\}$ and $f_I : \sigma'_{i_1} \times ... \times \sigma'_{i_k} \to \sigma'$, with $\sigma'_{i_j} \equiv \sigma_{i_j}$ for all $j \in \{1, ..., k\}$ and $\sigma' \equiv \sigma$. In order to include β-reduction we consider I to be the identity for λ and @. Additionally, we consider I to be the identity for variables (although it can be any bijection). We assume that there is only one pair for each symbol. Usually the identity pairs will be omitted. Thus the interpretation I of a term is obtained, as usual, by using the pairs on the top symbol once the arguments have been recursively interpreted.

Example 2 *Following the example 1, consider the interpretation I defined by the pairs $(map(x, y), y)$ and $(cons(x, y), cons_I(y))$, then we have that:*

1. $I(map(F, cons(x, xs))) = cons_I(xs)$
2. $I(cons(@(F, x), map(F, xs))) = cons_I(xs)$
3. $I(ps(cons(x, xs))) = ps(cons_I(xs))$
4. $I(cons(x, ps(map(\lambda y.x + y, xs)))) = cons_I(ps(xs))$
5. $I(@(F, x)) = @(F, x)$
6. $I(\lambda y.x + y) = \lambda y.x + y$

Now we show that this interpretations preserve typability, quasi-monotonicity, stability, α-compatibility and the inclusion of the β-reduction of $\succeq_B$.

Property 3 *Let I be an interpretation as defined above then we have the following preservation properties:*

1. *If s has type σ then $I(s)$ has a type τ such that $\tau \equiv \sigma$*
2. *If $\succeq_B$ is quasi-monotonic then $\succeq_I$ is quasi-monotonic.*
3. *If $\succeq_B$ is stable under substitutions then $\succeq_I$ is stable under substitutions.*
4. $\to_\beta \subseteq \succeq_B$ *implies* $\to_\beta \subseteq \succeq_I$

Proof. 1. By induction on $|s|$ and distinguishing cases according to the definition of the interpretations.

2. We have to prove that for all $s : \sigma$ and $t : \sigma'$ with $\sigma \equiv \sigma'$, $s \succeq_I t$ implies $f(...s...) \succeq_I f(...t...)$, $@(s,u) \succeq_I @(t,u)$, $@(u,s) \succeq_I @(u,t)$ and $\lambda x.s \succeq_I \lambda x.t$, for all $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $f \in \mathcal{F}$ and for $f(...w...)$, $@(w,u)$, $@(u,w)$, $\lambda x.w$ for $w \in \{s,t\}$ well-typed terms. By definition of $\succeq_I$, this is equivalent to prove first that $I(s) \succeq_B I(t)$ implies $I(f(...,s,...)) \succeq_B I(f(...,t,...))$, $I(@(s,u)) \succeq_B I(@(t,u))$, $I(@(u,s)) \succeq_B I(@(u,t))$ and $I(\lambda x.s) \succeq_B I(\lambda x.t)$, and second that all these terms are well-typed what we know it is true by property 3.1.
   - Consider first $f(...,s,...) : \tau \succeq_I f(...,t,...) : \tau'$ $(\tau \equiv \tau')$ for all $f \in \mathcal{F}$. The proof is very similar to the one done in [BFR00] for the first order case. It is done by analyzing the different kinds of interpretation we may have for I and using the quasi-monotonicity of $\succeq_B$.
   - $I(@(s,u)) = @(I(s), I(u))$ and $I(@(t,u)) = @(I(t), I(u))$.
     By quasi-monotonicity of $\succeq_B$, $I(s) \succeq_B I(t)$ implies
     $@(I(s), I(u)) \succeq_B @(I(t), I(u))$ and $@(I(u), I(s)) \succeq_B @(I(u), I(t))$.
   - $I(\lambda x.s) = \lambda x.I(s)$ and $I(\lambda x.t) = \lambda x.I(t)$. By quasi-monotonicity of $\succeq_B$, $I(s) \succeq_B I(t)$ implies $\lambda x.I(s) \succeq_B \lambda x.I(t)$.
3. To prove stability of $\succeq_I$, we define the interpretation of a substitution γ from variables of $\mathcal{X}$ to candidate terms as $I(\gamma) = \{I(x_i : \sigma_i) \mapsto I(x_i\gamma) : \sigma'_i \mid x_i \in Dom(\gamma) \textit{ and } \sigma_i \equiv \sigma'_i\}$. Then we can prove that $I(s\gamma) = I(s)I(\gamma)$ for all term $s : \sigma \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and for every substitution γ of typed terms for the variables of s, by induction on $|s|$ and distinguishing cases depending on whether s is a variable, a term $f(s_1...s_n)$, an application $@(u,v)$ or an abstraction $\lambda x.u$. Finally, by applying the definition of $\succeq_I$ and using the property, $I(s\gamma) = I(s)I(\gamma)$, we can easily prove that $\succ_B$ ($\succeq_B$, respectively) stable under substitutions implies $\succ_I$ ($\succeq_I$, respectively) stable under substitutions.
4. $\rightarrow_\beta \subseteq \succeq_I$ can be proved by using the definition of $\succeq_I$ and then that $\rightarrow_\beta \subseteq \succeq_B$.

5.2 Building $\succeq_Q$

Now we show how higher-order quasi-reduction pairs $\langle \succeq_I, \succeq_Q \rangle$ can be obtained. First we will just provide the most usual examples of quasi-orderings $\succeq_Q$ that work in practice.

The simplest case is consider that $\succeq_Q$ and $\succeq_I$ coincide, provided that $\succeq_I$ is a quasi-reduction ordering (which is the case with the $\succeq_I$ we have defined in the previous section).

A more elaborated case for $\succeq_Q$ is to combine lexicographically a precedence and the quasi-ordering $\succeq_I$, that is $\succeq_Q$ is defined as

$$(\succeq_{\mathcal{P}}, \succeq_I)_{lex}$$

where $\succeq_{\mathcal{P}}$ is a (well-founded) precedence, which can be completely different from the precedence used in the HORPO inside $\succeq_I$. Note that the precedence is used on terms by just comparing their head symbols. We can also have a lexicographic combination using first $\succeq_I$ and second the precedence.In both cases, since $\succeq_I$ has been built to be well-founded and the precedence is well-founded, its lexicographic combination also is. Quasi-monotonicity of $\succeq_I$ on $\succeq_Q$ can be easily shown using the quasi-monotonicity of $\succeq_I$.

Finally, using an idea coming from the dependency pair method [AG00], instead of using directly $\succeq_I$ inside $\succeq_Q$, we can apply first a *renaming* of the head symbol, in case it is a function symbol, before using $\succeq_I$. This renaming allows us to apply different interpretations when a function symbol occurs on the head than when it occurs inside the term. Therefore thanks to the renaming sometimes the proof can be easier.

Now we show how to obtain higher-order quasi-reduction pairs $\langle \succeq_I, \succeq_Q \rangle$ in a general way. This is done exactly as for MSPO in [BFR00].

First, as basic cases, we present two possible quasi-orderings $\succeq_Q$ which form higher-order quasi-reduction pair for a given higher-order quasi-reduction ordering $\succeq_I$.

- Identity: $\langle \succeq_I, \succeq_I \rangle$ is a quasi-reduction pair.
- Precedence: Let $\succeq_{\mathcal{F}}$ be a precedence on $\mathcal{F}$ and let $\succeq_T^{\mathcal{F}}$ be defined as $s \succeq_T^{\mathcal{F}} t$ iff $top(s) \succeq_{\mathcal{F}} top(t)$. Then $\langle \succeq_I, \succeq_T^{\mathcal{F}} \rangle$ is a quasi-reduction pair. In what follows we will write $\langle \succeq_I, \succeq_{\mathcal{F}} \rangle$ instead to denote the pair $\langle \succeq_I, \succeq_T^{\mathcal{F}} \rangle$.

Now we show how to obtain new higher-order quasi-reduction pairs from one or several given such pairs. Hence, we can start by pairs as in the two cases above and then (repeatedly) apply the following properties to obtain more suitable quasi-reduction pairs.

$\langle \succeq_I, \succeq_Q \rangle$ is a higher-order quasi-reduction pair if

- Extension: $\langle \succeq_I, \succeq_{Q_0} \rangle$ is a higher-order quasi-reduction pair and $\succeq_Q$ is well-founded and stable and $\succeq_{Q_0} \subseteq \succeq_Q$; or
- Renaming: $\langle \succeq_I, \succeq_{Q_0} \rangle$ is a quasi-reduction pair and $\succeq_Q$ is defined as $s \succeq_Q t$ if and only if either $s = t \in \mathcal{X}$ or $N(s) \succeq_{Q_0} N(t)$, where N is a *renaming* map defined as $N(f(t_1, \ldots, t_m)) = f_N(t_1, \ldots, t_m)$ for every symbol f in $\mathcal{F}$ and $N(t) = t$ if t is not a variable and it is not headed by a symbol in $\mathcal{F}$.
- Lexicographic combination: $\langle \succeq_I, \succeq_{Q_i} \rangle$ is a higher-order quasi-reduction pair for all $i \in \{1 \ldots n\}$ and $\succeq_Q$ is $(\succeq_{Q_1}, \ldots, \succeq_{Q_n})_{lex}$.

6 Examples

Let us illustrate the use and the power of MHOSPO, by means of several examples.

Example 3 *We recall example 1.*

$$map(F, []) \to [] \tag{1}$$
$$map(F, cons(x, xs)) \to cons(@(F, x), map(F, xs)) \tag{2}$$

$$ps([]) \to [] \tag{3}$$
$$ps(cons(x, xs)) \to cons(x, ps(map(\lambda y.x + y, xs))) \tag{4}$$

Termination of this TRS can be proved with $\succ_{mhospo}$ taking $\succeq_Q$ as $(\succeq_{\mathcal{P}}, \succeq_I)_{lex}$. The precedence $\succeq_{\mathcal{P}}$ is defined by $ps \succ_{\mathcal{P}} map \succ_{\mathcal{P}} cons$ and $ps \succ_{\mathcal{P}} +$; and we define $\succeq_I$ by combining $\succ_{horpo}$, with the precedence $ps \succ_{\mathcal{F}} cons_I$, and the interpretation function defined by the pairs $(map(x_1, x_2), x_2)$ and $(cons(x_1, x_2), cons_I(x_2))$, where $cons_I : List \to List$. Note that with this interpretation for the map function only the List parameter is considered and the List is interpreted as the amount of cons it has, which represents its size.

In order to prove that all rules are included in the obtained MHOSPO, we need to check both that $l \succeq_I r$ and $l \succ_{hospo} r$ for all rules $l \to r$ in the system. First of all recall that, since List and Nat are type variables (considered as sorts) we have $List \equiv Nat$, which will be used many times along the checking. We start showing that all rules are included in $\succeq_I$. Rules 1 and 2 have the same interpretation for both sides, $[]$ for the rule 1 and $cons_I(xs)$ for the rule 2. For rule 3 we trivially have $I(ps([])) = ps([]) \succ_{horpo} [] = I([])$ by case 1 of HORPO. For rule 4, $I(ps(cons(x, xs))) = ps(cons_I(xs)) \succ_{horpo} cons_I(ps(xs)) = I(cons(x, ps(map(\lambda y.x + y, xs))))$ by case 2 of HORPO since $ps \succ_{\mathcal{F}} cons_I$ and $ps(cons_I(xs)) \succ_{horpo} ps(xs)$ by using first the case 3 and then case 1.

Rules 1 and 3 are included in HOSPO by case 1. For rule 2 we show that $map(F, cons(x, xs)) \succ_{hospo} cons(@(F, x), map(F, xs))$. We apply first case 2, since $map(F, cons(x, xs)) \succ_Q cons(@(F, x), map(F, xs))$ by the first component of $\succeq_Q$ as $map \succ_{\mathcal{P}} cons$. Then we need to check recursively that (1) $map(F, cons(x, xs)) \succ_{hospo} @(F, x)$ and (2) $map(F, cons(x, xs)) \succ_{hospo} map(F, xs)$. For (1) we apply case 4 since $F \in \mathcal{CC}(map(F, cons(x, xs)))$ and $map(F, cons(x, xs)) \succ_{hospo} x$ applying case 1 twice. For (2) we apply again case 2 since $map(F, cons(x, xs)) \succ_Q map(F, xs)$, by the second component of $\succeq_Q$ as $I(map(F, cons(x, xs))) = cons_I(xs) \succ_{horpo} xs = I(map(F, xs))$, and the recursive checking hold in the same way as before.

For the last rule we need $ps(cons(x, xs)) \succ_{hospo} cons(x, ps(map(\lambda y.x + y, xs)))$. We apply case 2 using the precedence component of $\succeq_Q$. For the recursive checking, $ps(cons(x, xs)) \succ_{hospo} x$ is proved by applying case 1 twice, and $ps(cons(x, xs)) \succ_{hospo} ps(map(\lambda y.x+y, xs))$ is proved by case 2 since $ps(cons(x, xs)) \succ_Q ps(map(\lambda y.x+y, xs))$ by the second component of $\succeq_Q$, as $I(ps(cons(x, xs)) = ps(cons_I(xs)) \succ_{horpo} ps(xs) = I(ps(map(\lambda y.x + y, xs)))$, and to prove $ps(cons(x, xs)) \succ_{hospo} map(\lambda y.x + y, xs)$ we can use again case 2, in this case using the precedence component of $\succeq_Q$ since $ps \succ_{\mathcal{P}} map$. Finally, for the recursive checking we have that $ps(cons(x, xs)) \succ_{hospo} xs$ by case 1 and for the first argument we will show that (a) $\lambda y.cons(x, xs) + y \in \mathcal{CC}(ps(cons(x, xs)))$ and (b) that $\lambda y.cons(x, xs) + y \succ_{hospo} \lambda y.x + y$. For (a) we use case 2 of the computable closure, and hence we have to prove that $cons(x, xs) + y \in \mathcal{CC}(ps(cons(x, xs)), \{y\})$ which holds by using case 1 of the computable closure since $cons(x, xs) \in \mathcal{CC}(ps(cons(x, xs)), \{y\})$, $y \in \mathcal{CC}(ps(cons(x, xs)), \{y\})$ and $ps(cons(x, xs)) \succ_Q cons(x, xs)+y$ since $ps \succ_{\mathcal{P}} +$ (note that $\lambda y.cons(x, xs) + y$ is well-typed since $Nat \equiv List$). Finally for (b) we can use case 6 and then to prove $cons(x, xs)+y \succ_{hospo} x+y$ we can apply case 2 by using the second component of $\succeq_Q$ as $I(cons(x, xs)+y) = cons_I(xs)+y \succ_{horpo} x+y = I(x+y)$ by case 3 first, and then case 1, and for the recursive checking $cons(x, xs)+y \succ_{hospo}$

x is proved by applying case 1 twice and $cons(x, xs) + y \succ_{hospo} y$ is proved again by case 1.

The last part of the proof of the previous example may look hard to automate, since we have to pick a term from the computable closure of $ps(cons(x, xs))$ and then check that it is greater than or equal to $\lambda y.x + y$. In practice, we look for such a term in the computable closure in a goal-directed way: since $\lambda y.x + y$ is headed by lambda and then $+$ we apply case 2 and case 1, and then, when we reach x, we check whether some argument of $ps(cons(x, xs))$ (which belongs to the computable closure by definition) is greater than or equal to x. Therefore, since $cons(x, xs) \succ_{hospo} x$, by monotonicity we conclude, without any additional checking, that $\lambda y.cons(x, xs) + y \succ_{hospo} \lambda y.x + y$

Let us present another example which can be proved by MHOSPO and not by HORPO.

Example 4 *Quick sort Let $V_{\mathbb{T}} = \{Bool, Nat, List\}$,*
$\mathcal{X} = \{x, y : Nat; xs, ys : List; p : Nat \to Bool\}$ *and*
$\mathcal{F} = \{\ 0 : Nat,\ s : Nat \to Nat,\ le, gr : Nat \times Nat \to Bool,$
$True, False : Bool,\ if : Bool \times List \times List \to List,$
$[] : List,\ cons : Nat \times List \to List,\ ++ : List \times List \to List,$
$filter : (Nat \to Bool) \times List \to List,\ qsort : List \to List \quad \}$

$$\begin{array}{ll} if(True, xs, ys) \to xs & [] ++ xs \to xs \\ if(False, xs, ys) \to ys & cons(x, xs) ++ ys \to cons(x, xs ++ ys) \end{array}$$

$$\begin{array}{ll} le(0, y) \to True & gr(0, y) \to False \\ le(s(x), 0) \to False & gr(s(x), 0) \to True \\ le(s(x), s(y)) \to le(x, y) & gr(s(x), s(y)) \to gr(x, y) \end{array}$$

$$\begin{array}{l} filter(p, []) \to [] \\ filter(p, cons(x, xs)) \to if(@(p, x), cons(x, filter(p, xs)), filter(p, xs)) \end{array}$$

$$\begin{array}{l} qsort([]) \to [] \\ qsort(cons(x, xs)) \to qsort(filter(\lambda z.le(z, x), xs)) ++ cons(x, []) ++ \\ \qquad\qquad qsort(filter(\lambda z.gr(z, x), xs)) \end{array}$$

Termination of this TRS can be proved with $\succ_{mhospo}$ taking $\succeq_Q$ as $(\succeq_{\mathcal{P}}, \succeq_I)_{lex}$. The precedence $\succeq_{\mathcal{P}}$ is defined by $0 \succ_{\mathcal{P}} True, False$; $qsort \succ_{\mathcal{P}} filter, ++, cons, le, gr, []$ and $++ \succ_{\mathcal{P}} cons =_{\mathcal{P}} filter \succ_{\mathcal{P}} if$, and we define $\succeq_I$ by combining $\succ_{horpo}$ as basic ordering and the interpretation I defined by the pairs $(filter(x_1, x_2), filter_I(x_2))$ and $(if(x_1, x_2, x_3), if_I(x_2, x_3))$ where $filter_I : List \to List$ and $if_I : List \times List \to List$. As precedence for $\succ_{horpo}$ we take $0 \succ_{\mathcal{F}} True, False$; $qsort \succ_{\mathcal{F}} ++, cons, []$ and $++ \succ_{\mathcal{F}} cons =_{\mathcal{F}} filter_I \succ_{\mathcal{F}} if_I$.

In order to prove that all rules are included in the obtained MHOSPO, we need to check both that $l \succeq_I r$ and $l \succ_{hospo} r$ for all rules $l \to r$ in the system.

First we show that all rules are included in $\succeq_I$, that is, $I(l) \succeq_{horpo} I(r)$ for all rule $l \to r$.

- $if(True, x, y) \succ_I x$ since $I(if(True, x, y)) = if_I(x, y)$, $I(x) = x$ and $if_I(x, y) \succ_{horpo} x$ by case 1.
- $if(False, x, y) \succ_I y$ since $I(if(False, x, y)) = if_I(x, y)$, $I(y) = y$ and $if_I(x, y) \succ_{horpo} y$ by case 1.
- $[] +\!\!+ y \succ_I y$ since $I([] +\!\!+ y) = [] +\!\!+ y$, $I(y) = y$ and $[] +\!\!+ y \succ_{horpo} y$ by case 1.
- $cons(x, z) +\!\!+ y \succ_I cons(x, z +\!\!+ y)$ since $I(cons(x, z) +\!\!+ y) = cons(x, z) +\!\!+ y$, $I(cons(x, z +\!\!+ y)) = cons(x, z +\!\!+ y)$ and $cons(x, z) +\!\!+ y \succ_{horpo} cons(x, z +\!\!+ y)$ by case 2 since $+\!\!+ \succ_{\mathcal{F}} cons$ and $cons(x, z) +\!\!+ y \succ_{horpo} x$ by case 1 and $cons(x, z) +\!\!+ y \succ_{horpo} z +\!\!+ y$ by case 3 since $cons(x, z) \succ_{horpo} z$ by case 1 and $y \succeq_{horpo} y$.
- $le(0, y) \succ_I True$ since $I(le(0, y)) = le(0, y)$, $I(True) = True$ and $le(0, y) \succ_{horpo} True$ by case 1 since $0 \succ_{\mathcal{F}} True$ implies $0 \succ_{horpo} True$ by case 2.
- $le(s(x), 0) \succ_I False$ since $I(le(s(x), 0)) = le(s(x), 0)$, $I(False) = False$ and $le(s(x), 0) \succ_{horpo} False$ by case 1 since $0 \succ_{\mathcal{F}} False$ implies $0 \succ_{horpo} False$ by case 2.
- $le(s(x), s(y)) \succ_I le(x, y)$ since $I(le(s(x), s(y))) = le(s(x), s(y))$, $I(le(x, y)) = le(x, y)$ and $le(s(x), s(y)) \succ_{horpo} le(x, y)$ by case 3.
- $gr(0, y) \succ_I False$, $gr(s(x), 0) \succ_I True$ and $gr(s(x), s(y)) \succ_I gr(x, y)$ are proved exactly as done for the rules which define le.
- $filter(p, []) \succ_I []$ since $I(filter(p, [])) = filter_I([])$, $I([]) = []$ and $filter_I([]) \succ_{horpo} []$ by case 1.
- $filter(p, cons(x, xs)) \succ_I if(@(p, x), cons(x, filter(p, xs)), filter(p, xs))$ since $I(filter(p, cons(x, xs))) = filter_I(cons(x, xs))$,
  $I(\ if(@(p, x)\ ,\ cons(x, filter(p, xs))\ ,\ filter(p, xs)))\ =$
  $if_I(cons(x, filter_I(xs))\ ,\ filter_I(xs)))$ and
  $filter_I(cons(x, xs)) \succ_{horpo} if_I(cons(x, filter_I(xs)), filter_I(xs))$ by case 2:
  - $filter_I \succ_{\mathcal{F}} if_I$.
  - $filter_I(cons(x, xs)) \succ_{horpo} cons(x, filter_I(xs))$ by case 3: $filter_I =_{\mathcal{F}} cons$ and $\{cons(x, xs)\} \succ\!\!\succ_{horpo} \{x, filter_I(xs)\}$ since $cons(x, xs)) \succ_{horpo} x$ by case 1 and $cons(x, xs)) \succ_{horpo} filter_I(xs)$ by case 3 since $filter_I =_{\mathcal{F}} cons$ and $\{x, xs\} \succ\!\!\succ_{horpo} \{xs\}$.
  - $filter_I(cons(x, xs)) \succ_{horpo} filter_I(xs)$ by case 3.
- $qsort([]) \succ_I []$ since $I(qsort([])) = qsort([])$, $I([]) = []$ and $qsort([]) \succ_{horpo} []$ by case 1.
- $qsort(cons(x, xs)) \succ_I qsort(filter(\lambda y.le(y, x), xs)) +\!\!+ cons(x, []) +\!\!+ qsort(filter(\lambda y.gr(y, x), xs))$ since $I(qsort(cons(x, xs))) = qsort(cons(x, xs))$, $I(qsort(filter(\lambda y.le(y, x), xs)) +\!\!+ cons(x, []) +\!\!+ qsort(filter(\lambda y.gr(y, x), xs))) = qsort(filter_I(xs)) +\!\!+ cons(x, []) +\!\!+ qsort(filter_I(xs))$ and
  $qsort(cons(x, xs)) \succ_{horpo} qsort(filter_I(xs)) +\!\!+ cons(x, []) +\!\!+ qsort(filter_I(xs))$ by case 2 of HORPO:
  - $qsort \succ_{\mathcal{F}} +\!\!+$
  - $qsort(cons(x, xs)) \succ_{horpo} cons(x, [])$ by case 2 since $qsort \succ_{\mathcal{F}} cons$, $qsort(cons(x, xs)) \succ_{horpo} x$ by case 1 and $qsort \succ_{\mathcal{F}} []$ implies $qsort(cons(x, xs)) \succ_{horpo} []$ by case 2.
  - $qsort(cons(x, xs)) \succ_{horpo} qsort(filter_I(xs))$ by applying twice case 3 since $cons =_{\mathcal{F}} filter_I$ and $\{x, xs\} \succ\!\!\succ_{horpo} \{xs\}$.

Now we prove that all rules are included in $\succ_{hospo}$.

- $if(True, x, y) \succ_{hospo} x$ *by case 1.*
- $if(False, x, y) \succ_{hospo} y$ *by case 1.*
- $[] +\!\!+ y \succ_{hospo} y$ *by case 1.*
- $cons(x, z) +\!\!+ y \succ_{hospo} cons(x, z +\!\!+ y)$ *by case 2 since,* $+\!\!+ \succ_{\mathcal{P}} cons$*, implies* $cons(x, z) +\!\!+ y \succ_{Q} cons(x, z +\!\!+ y)$ *and* $cons(x, z) +\!\!+ y \succ_{hospo} x$ *by case 1 and* $cons(x, z) +\!\!+ y \succ_{hospo} z +\!\!+ y$ *by case 2 since* $cons(x, z) +\!\!+ y \succ_{I} z +\!\!+ y$*, and* $cons(x, z) +\!\!+ y \succ_{hospo} z$ *and* $cons(x, z) +\!\!+ y \succ_{hospo} y$ *by case 1.*
- $le(0, y) \succ_{hospo} True$ *by case 1 since* $0 \succ_{\mathcal{P}} True$ *implies* $0 \succ_{hospo} True$ *by case 2.*
- $le(s(x), 0) \succ_{hospo} False$ *by case 1 since* $0 \succ_{\mathcal{P}} False$ *implies* $0 \succ_{hospo} False$ *by case 2.*
- $le(s(x), s(y)) \succ_{hospo} le(x, y)$ *by case 2 since, as seen above,* $le(s(x), s(y)) \succ_{I} le(x, y)$*, and* $le(s(x), s(y)) \succ_{hospo} x$ *and* $le(s(x), s(y)) \succ_{hospo} y$ *by case 1.*
- $gr(0, y) \succ_{hospo} False$*,* $gr(s(x), 0) \succ_{hospo} True$ *and* $gr(s(x), s(y)) \succ_{hospo} gr(x, y)$ *are proved exactly as done for the rules which define le.*
- $if(p, []) \succ_{hospo} []$ *by case 1.*
- $filter(p, cons(x, xs)) \succ_{hospo} if(@(p, x), cons(x, filter(p, xs)), filter(p, xs))$ *by case 2:*
  - $filter(\ p, cons(x, xs)\ ) \succ_{Q} if(\ @(p, x), cons(x, filter(p, xs)), filter(p, xs)\ )$ *since* $filter \succ_{\mathcal{P}} if$.
  - $filter(p, cons(x, xs)) \succ_{hospo} @(p, x)$ *by case 4.*
  - $filter(p, cons(x, xs)) \succ_{hospo} cons(x, filter(p, xs))$ *by case 2:*
    - $filter(p, cons(x, xs)) \succ_{Q} cons(x, filter(p, xs))$ *since* $filter =_{\mathcal{P}} cons$ *and* $filter(p, cons(x, xs)) \succ_{I} cons(x, filter(p, xs))$ *since* $I(filter(p, cons(x, xs)) = filter_I(cons(x, xs))$, $I(cons(x, filter(p, xs)) = cons(x, filter_I(xs))$ *and* $filter_I(cons(x, xs)) \succ_{horpo} cons(x, filter_I(xs))$ *since* $filter_I =_{\mathcal{F}} cons$ *and* $\{cons(x, xs)\} \succ\!\!\succ_{horpo} \{x, filter_I(xs)\}$ *(as seen above).*
    - $filter(p, cons(x, xs)) \succ_{hospo} x$ *by case 1.*
    - $filter(p, cons(x, xs)) \succ_{hospo} filter(p, xs)$ *by case 2:* $filter(p, cons(x, xs)) \succ_{I} filter(p, xs)$ *since* $I(filter(p, cons(x, xs))) = filter_I(cons(x, xs)) \succ_{horpo} filter_I(xs) = I(filter(p, xs))$ *and* $p \succeq_{hospo} p$*, and* $filter(p, cons(x, xs)) \succ_{hospo} xs$ *by case 1.*
  - $filter(p, cons(x, xs)) \succ_{hospo} filter(p, xs)$ *by case 2, as seen in the previous case.*
- $qsort([]) \succ_{hospo} []$ *by case 1.*
- $qsort(cons(x, xs)) \succ_{hospo} qsort(filter((\lambda y.le(y, x)), xs)) +\!\!+ cons(x, []) +\!\!+ qsort(filter((\lambda y.gr(y, x)), xs))$ *by case 2:*
  - $qsort(x : xs) \succ_{Q} qsort(filter((\lambda y.le(y, x)), xs)) +\!\!+ cons(x, []) +\!\!+ qsort(filter((\lambda y.gr(y, x)), xs))$ *since,* $qsort \succ_{\mathcal{P}} +\!\!+$.
  - $qsort(cons(x, xs)) \succ_{hospo} qsort(filter((\lambda y.le(y, x)), xs))$ *by case 2:*
    - $qsort(cons(x, xs)) \succ_{Q} qsort(filter((\lambda y.le(y, x)), xs))$ *since* $I(qsort(cons(x, xs))) = qsort(cons(x, xs)) \succ_{horpo} qsort(filter_I(xs)) = I(qsort(filter((\lambda y.le(y, x)), xs))$ *by case 3 applied twice, since* $cons =_{\mathcal{F}} filter_I$ *and* $\{x, xs\} \succ\!\!\succ_{horpo} \{xs\}$.
    - $qsort(cons(x, xs)) \succ_{hospo} filter((\lambda y.le(y, x)), xs)$ *by case 2 since:*
      - $qsort(cons(x, xs)) \succ_{Q} filter((\lambda y.le(y, x)), xs)$ *since* $qsort \succ_{\mathcal{P}} filter$.

- $\lambda y.le(y, cons(x, xs)) \in \mathcal{CC}(qsort(cons(x, xs)))$ and we can prove $\lambda y.le(y, cons(x, xs)) \succ_{hospo} \lambda y.le(y, x)$ by using first case 6 and then case 2. To prove $\lambda y.le(y, cons(x, xs)) \in \mathcal{CC}(qsort(cons(x, xs)))$ we can use case 2 since $y \in \mathcal{CC}(qsort(cons(x, xs)), \{y\})$, $cons(x, xs) \in \mathcal{CC}(qsort(cons(x, xs)), \{y\})$ and $le(y, cons(x, xs)) \in \mathcal{CC}(qsort(cons(x, xs)), \{y\})$ can be proved by using case 1 since $qsort \succ_{\mathcal{P}} le$ implies $qsort(cons(x, xs)) \succ_Q le(y, cons(x, xs))$.
- $qsort(cons(x, xs)) \succ_{hospo} xs$ by case 1.

- $qsort(cons(x, xs)) \succ_{hospo} qsort(filter((\lambda y.gr(y, x)), xs))$ by case 2. It is proved as done in the previous case.
- $qsort(cons(x, xs)) \succ_{hospo} cons(x, [])$ by case 2 since $qsort \succ_{\mathcal{P}} cons$ implies $qsort(cons(x, xs)) \succ_Q cons(x, [])$ and $qsort(cons(x, xs)) \succ_{hospo} x$ by case 1 and $qsort(cons(x, xs)) \succ_{hospo} []$ by case 2 since $qsort \succ_{\mathcal{P}} []$.

The following example is a disjoint union of a first-order TRS, which can be proved terminating by MSPO (or by the dependency pair method), and a higher-order TRS, which can be proved terminating by HORPO, plus a higher-order rule which represents an initial call. Although its components can be proved terminating separately, no method can ensure the termination of the whole system. Using MHOSPO, we can somehow combine the proofs (used in MSPO and HORPO) and show its termination.

Example 5 *Let* $V_{\mathbb{T}} = \{Bool, Nat, List\}$,
$\mathcal{X} = \{x, y : Nat; xs, ys : List; f : Nat \to Nat \to Nat\}$ *and*
$\mathcal{F} = \{\ 0 : Nat,\ s : Nat \to Nat,\ le : Nat \times Nat \to Bool,$
$gcd, minus : Nat \times Nat \to Nat,$
$True, False : Bool,\ if : Bool \times Nat \times Nat \to Nat,$
$[] : List,\ cons : Nat \times List \to List,$
$zipWith : (Nat \to Nat \to Nat) \times List \times List \to List,$
$gcdlists : List \times List \to List\ \}$

$$\begin{array}{ll} le(0, y) \to True & \\ le(s(x), 0) \to False & minus(x, 0) \to x \\ le(s(x), s(y)) \to le(x, y) & minus(s(x), s(y)) \to minus(x, y) \end{array}$$

$$\begin{array}{ll} gcd(0, y) \to 0 & \\ gcd(s(x), 0) \to 0 & \\ gcd(s(x), s(y)) \to if(le(y, x), s(x), s(y)) & (1) \\ if(True, s(x), s(y)) \to gcd(minus(x, y), s(y)) & (2) \\ if(False, s(x), s(y)) \to gcd(minus(y, x), s(x)) & \end{array}$$

$$\begin{array}{ll} zipWith(f, xs, []) \to [] & \\ zipWith(f, [], ys) \to [] & \\ zipWith(f, cons(x, xs), cons(y, ys)) \to cons(@(f, x, y), zipWith(f, xs, ys)) & (3) \end{array}$$

$$gcdlists(xs, ys) \to zipWith(\lambda x, y.gcd(x, y), xs, ys) \quad (4)$$

Termination of this TRS can be proved with $\succ_{mhospo}$ *taking* $\succeq_Q$ *as* $(\succeq_{\mathcal{P}_1}, \succeq_I, \succeq_{\mathcal{P}_2})_{lex}$ *with* $\succeq_{\mathcal{P}_1}$ *s.t.* $le \succ_{\mathcal{P}_1} True, False$; $gcd =_{\mathcal{P}_1} if \succ_{\mathcal{P}_1} minus$; $gcd \succ_{\mathcal{P}_1} le$; $zipWith \succ_{\mathcal{P}_1} cons$ *and* $gcdlists \succ_{\mathcal{P}_1} zipWith, gcd$; *and* $\succeq_{\mathcal{P}_2}$ *s.t.* $gcd \succ_{\mathcal{P}_2} if$. *For* $\succeq_I$ *we combine HORPO as basic ordering and the interpretation function defined by the pairs* $(minus(x,y), minus_I(x))$, $(if(x,y,z), if_I(y,z))$ *where* $minus_I : Nat \to Nat$ *and* $if_I : Nat \times Nat \to Nat$. *As underlying precedence for HORPO we take* $\succeq_{\mathcal{F}}$ *s.t.* $s \succ_{\mathcal{F}} minus_I$, $if_I =_{\mathcal{F}} gcd$, $le \succ_{\mathcal{F}} True, False$, $gcdlists \succ_{\mathcal{F}} zipWith, gcd$ *and* $zipWith \succ_{\mathcal{F}} cons$.

In order to prove that all rules are included in the obtained MHOSPO, we need to check both that $l \succeq_I r$ *and* $l \succ_{hospo} r$ *for all rules* $l \to r$ *in the system.*

We only show in detail the enumerated rules. For the rest of the rules, to see that they are included in $\succ_{mhospo}$ *is trivial.*

First we show that the rules are included in $\succeq_I$.
For rule 1 *we have* $I(gcd(s(x), s(y))) = gcd(s(x), s(y)) =_{horpo} if_I(s(x), s(y)) = I(if(le(y,x), s(x), s(y)))$ *since* $gcd =_{\mathcal{F}} if_I$. *For rule* 2 $I(if(True, s(x), s(y))) = if_I(s(x), s(y)) \succ_{horpo} gcd(minus_I(x), s(y)) = I(gcd(minus(x,y), s(y)))$ *by case 3 of HORPO, since* $if_I =_{\mathcal{F}} gcd$ *and* $s \succ_{\mathcal{F}} minus_I$ *implies* $s(x) \succ_{horpo} minus_I(x)$ *by case 2 of HORPO. For rule* 3 $I(zipWith(f, cons(x, xs), cons(y, ys))) = zipWith(f, cons(x, xs), cons(y, ys)) \succ_{horpo} cons(@(f,x,y), zipWith(f, xs, ys)) = I(cons(@(f,x,y), zipWith(f, xs, ys)))$ *by case 2 of HORPO, since* $zipWith \succ_{\mathcal{F}} cons$, $zipWith(f, cons(x, xs), cons(y, ys)) \succ_{horpo} @(f,x,y)$ *by case 5 and* $zipWith(f, cons(x, xs), cons(y, ys)) \succ_{horpo} zipWith(f, xs, ys)$ *by case 3. For rule 4* $I(gcdlists(xs, ys)) = gcdlists(xs, ys) \succ_{horpo} zipWith(\lambda x, y.gcd(x,y), xs, ys) = I(zipWith(\lambda x, y.gcd(x,y), xs, ys))$, *by case 2 of HORPO since* $gcdlists \succ_{\mathcal{F}} zipWith$ *and* $gcdlists \succ_{\mathcal{F}} gcd$ *implies* $\lambda x, y.gcd(x,y) \in \mathcal{CC}(gcdlists(xs, ys))$ *and both* $gcdlists(xs, ys) \succ_{horpo} xs$ *and* $gcdlists(xs, ys) \succ_{horpo} ys$ *by case 1.*

Now we will show that the rules are included in HOSPO.

For rule 1, $gcd(s(x), s(y)) \succ_{hospo} if(le(y,x), s(x), s(y))$ *by case 2 since* $gcd(s(x), s(y)) \succ_Q if(le(y,x), s(x), s(y))$ *by the second precedence* $\succ_{\mathcal{P}_2}$ *since* $gcd =_{\mathcal{P}_1} if$, $gcd(s(x), s(y)) =_I if(le(y,x), s(x), s(y))$, *and* $gcd \succ_{\mathcal{P}_2} if$. *For the arguments, we can prove* $gcd(s(x), s(y)) \succ_{hospo} s(x)$ *and* $gcd(s(x), s(y)) \succ_{hospo} s(y)$ *by case 1 and* $gcd(s(x), s(y)) \succ_{hospo} le(y,x)$ *by case 2 since* $gcd \succ_{\mathcal{P}_1} le$ *implies* $gcd(s(x), s(y)) \succ_Q le(y,x)$ *and both* $gcd(s(x), s(y)) \succ_{hospo} y$ *and* $gcd(s(x), s(y)) \succ_{hospo} x$ *are proved by applying case 1 twice. For rule* 2, $if(True, s(x), s(y)) \succ_{hospo} gcd(minus(x,y), s(y))$ *by case 2 since* $gcd =_{\mathcal{P}_1} if$ *and* $if(True, s(x), s(y)) \succ_I gcd(minus(x,y), s(y))$ *(seen above) imply* $if(True, s(x), s(y)) \succ_Q gcd(minus(x,y), s(y))$, *and for the arguments* $if(True, s(x), s(y)) \succ_{hospo} s(y)$ *by case 1 and* $if(True, s(x), s(y)) \succ_{hospo} minus(x,y)$ *by case 2 since* $if(True, s(x), s(y)) \succ_Q minus(x,y)$ *by the precedence* $\succ_{\mathcal{P}_1}$ *and both* $if(True, s(x), s(y)) \succ_{hospo} x$ *and* $if(True, s(x), s(y)) \succ_{hospo} y$ *can be proved by applying case 1 twice.*
For rule 3 $zipWith(f, x : xs, y : ys) \succ_{hospo} cons(@(f,x,y), zipWith(f, xs, ys))$ *by case 2 since* $zipWith(f, cons(x, xs), cons(y, ys)) \succ_Q cons(@(f,x,y), zipWith(f, xs, ys))$ *by using* $\succ_{\mathcal{P}_1}$, *and* $zipWith(f, cons(x, xs), cons(y, ys)) \succ_{hospo} @(f,x,y)$ *by case 4 and* $zipWith(f, cons(x, xs), cons(y, ys)) \succ_{hospo} zipWith(f, xs, ys)$ *by case 2 by using* $\succ_I$ *(seen above) and case 1 for the arguments.*

For rule 4 $gcdlists(xs, ys) \succ_{hospo} zipWith((\lambda x, y.gcd(x, y)), xs, ys)$ *by case 2 since* $gcdlists(xs, ys) \succ_Q zipWith((\lambda x, y.gcd(x, y)), xs, ys)$ *using* $gcdlists \succ_{\mathcal{P}_1} zipWith$ *and finally, both* $gcdlists(xs, ys) \succ_{hospo} xs$ *and* $gcdlists(xs, ys) \succ_{hospo} ys$ *hold by case 1 and* $\lambda x, y.gcd(x, y) \in \mathcal{CC}(gcdlists(xs, ys))$ *since* $gcdlist \succ_{\mathcal{P}_1} gcd$ *implies* $gcdlists(xs, ys) \succ_Q gcd(x, y)$.

7 Improving the Computable Closure

In this section we extend the definition of the computable closure with some computability preserving operations given in [JR99].

Definition 8. *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* $s : \tau \in \mathcal{CC}(t, \mathcal{V})$ *and* $u : \sigma$ *be a subterm of* s *(through equivalent types) such that* $\tau \equiv \sigma$ *and* $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$; *then* $u \in \mathcal{CC}(t, \mathcal{V})$.
2. *quasi-ordering: Let* $g(\overline{s})$ *such that* $f(\overline{t}) \succ_Q g(\overline{s})$ *and* $\overline{s} \in \mathcal{CC}(t, \mathcal{V})$; *then* $g(\overline{s}) \in \mathcal{CC}(t, \mathcal{V})$.
3. *application: Let* $s : \sigma_1 \to \sigma_2 \to \dots \to \sigma_n \to \sigma \in \mathcal{CC}(t, \mathcal{V})$ *and* $u_i : \sigma_i \in \mathcal{CC}(t, \mathcal{V})$ *for every* $i \in \{1...n\}$; *then* $@(s, u_1, ..., u_n) \in \mathcal{CC}(t, \mathcal{V})$.
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* $u \to_\beta v$; *then* $v \in \mathcal{CC}(t, \mathcal{V})$.

The following lemma is necessary to proof stability under substitutions of $\succ_{hospo}$.

Lemma 6. *Assume that* $u \in \mathcal{CC}(t)$. *Then,* $u\gamma \in \mathcal{CC}(t\gamma)$ *for every substitution* γ.

Proof. We prove that if $u \in \mathcal{CC}(t, \mathcal{V})$ with $\mathcal{V} \subseteq \mathcal{X} \setminus (\mathcal{V}ar(t) \cup \mathcal{V}ar(t\gamma) \cup Dom(\gamma))$, then $u\gamma \in \mathcal{CC}(t\gamma, \mathcal{V})$. By induction on the definition of $\mathcal{CC}(t, \mathcal{V})$. We proceed by distinguishing cases according to the definition of the computational closure.

1. Let $u \in \mathcal{CC}(t, \mathcal{V})$ by case 1, that is, $s : \tau \in \mathcal{CC}(t, \mathcal{V})$ and $u : \sigma$ be a subterm of s with $\tau \equiv \sigma$ and $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$. By induction hypothesis we have $s\gamma : \tau \in \mathcal{CC}(t\gamma, \mathcal{V})$, and since $s\gamma \rhd u\gamma$ and $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$ implies $\mathcal{V}ar(u\gamma) \subseteq \mathcal{V}ar(t\gamma)$ then $u\gamma \in \mathcal{CC}(t\gamma, \mathcal{V})$ by case 1.
2. Let $u \in \mathcal{CC}(t, \mathcal{V})$ by case 2, that is, $u = g(\overline{s})$ such that $f(\overline{t}) \succ_Q g(\overline{s})$ and $\overline{s} \in \mathcal{CC}(t, \mathcal{V})$. By induction hypothesis $\overline{s}\gamma \in \mathcal{CC}(t\gamma, \mathcal{V})$, and by stability under substitutions of $\succ_Q$, $f(\overline{t})\gamma \succ_Q g(\overline{s})\gamma = g(\overline{s}\gamma)$. Therefore, $g(\overline{s})\gamma \in \mathcal{CC}(t\gamma, \mathcal{V})$ by case 2.
3. Let $u = @(s, u_1, ..., u_n) \in \mathcal{CC}(t, \mathcal{V})$ by case 3, that is, $s : \sigma_1 \to \sigma_2 \to \dots \to \sigma_n \to \sigma \in \mathcal{CC}(t, \mathcal{V})$ and $u_i : \sigma_i \in \mathcal{CC}(t, \mathcal{V})$ for every $i \in \{1...n\}$. By induction hypothesis, $s\gamma : \sigma_1 \to \sigma_2 \to \dots \to \sigma_n \to \sigma \in \mathcal{CC}(t\gamma, \mathcal{V})$ and $u_i\gamma : \sigma_i \in \mathcal{CC}(t\gamma, \mathcal{V})$ for every $i \in \{1...n\}$. Therefore, $@(s\gamma, u_1\gamma, ..., u_n\gamma) = u\gamma \in \mathcal{CC}(t\gamma, \mathcal{V})$.

4. Let $u = \lambda x.s \in \mathcal{CC}(t, \mathcal{V})$ by case 4, that is, $x \notin \mathcal{V}ar(t) \cup \mathcal{V}$ and $s \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$. To the price of renaming x if necessary, we can assume in addition that $x \notin \mathcal{V}ar(t\gamma) \cup \mathcal{D}om(\gamma)$, and therefore, $x \in \mathcal{X} \setminus (\mathcal{V} \cup \mathcal{V}ar(t) \cup \mathcal{V}ar(t\gamma) \cup \mathcal{D}om(\gamma))$. By induction hypothesis, $s\gamma \in \mathcal{CC}(t\gamma, \mathcal{V} \cup \{x\})$, and since $x \notin \mathcal{V}ar(t\gamma) \cup \mathcal{V}$ then $\lambda x.s\gamma \in \mathcal{CC}(t\gamma, \mathcal{V})$. Using now the hypothesis that $x \notin \mathcal{D}om(\gamma)$, $u\gamma = \lambda x.s\gamma$, and we are done.
5. Let $u \in \mathcal{CC}(t, \mathcal{V})$ by case 5, that is, $w \in \mathcal{CC}(t, \mathcal{V})$, and $w \rightarrow_\beta u$. By induction hypothesis $w\gamma \in \mathcal{CC}(t\gamma, \mathcal{V})$, and since, by stability of the beta-reduction $w\gamma \rightarrow_\beta u\gamma$ then $u\gamma \in \mathcal{CC}(t\gamma, \mathcal{V})$ by case 5.

To prove the well-foundedness of $\succ_{hospo}$ we need to prove 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 *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 $\langle \succ_Q, \succ\!\!\succ_{hospo} \rangle$ operating on pairs $\langle f(\overline{t}), \overline{t} \rangle$. Then every term in $\mathcal{CC}(t)$ is computable.*

Proof. We prove $u\gamma$ 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 $\mathcal{V} = \emptyset$. We proceed by induction on the definition of $\mathcal{CC}(t, \mathcal{V})$.

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.

For the induction step, we distinguish cases according to the definition of the computable closure:

1. Let $u : \sigma$ be a subterm of $s : \rho \in \mathcal{CC}(t, \mathcal{V})$ (through equivalent types), $\rho \equiv \sigma$, $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$. Therefore, $s \succ_{hospo} u$ by case 1 (since we reach u from s through equivalent types). By induction hypothesis $s\gamma$ is computable and since, by stability of $\succ_{hospo}$, $s\gamma \succ_{hospo} u\gamma$ then $u\gamma$ is computable by property 1.2.
2. case 2: $u = g(\overline{s})$, $\overline{s} \in \mathcal{CC}(t, \mathcal{V})$ and $f(\overline{t}) \succ_Q g(\overline{s})$. By induction hypothesis, $\overline{s}\gamma$ is computable and since, by stability of $\succ_Q$, $f(\overline{t})\gamma \succ_Q g(\overline{s})\gamma$, then $u\gamma$ is computable by assumption.
3. case 3: by induction and property 1.4.
4. case 4: let $u = \lambda x.s$ with $x \notin \mathcal{V} \cup \mathcal{V}ar(t)$ and $s \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$. To the price of possibly renaming x, we can assume without loss of generality that $x \notin \mathcal{D}om(\gamma)$. Therefore $\mathcal{V} \cup \{x\} \cap \mathcal{V}ar(t) = \emptyset$, and given an arbitrary computable term w, $\gamma' = \gamma \cup \{x \mapsto w\}$ is a computable substitution of domain $\mathcal{V} \cup \{x\}$. By induction hypothesis, $s\gamma'$ is therefore computable, and by property 1.5 $(\lambda x.s)\gamma$ is computable.
5. case 5: by induction, stability of $\rightarrow_\beta$ and property 1.2.

Now lemma 4 can be proved as before but using property 4 instead of property 2.

8 Constraints

We are currently working on a constraint-based termination proof method, in the light of the dependency pair method [AG00], for the higher-order case. Using the ideas of [BFR00], the constraints are extracted from the definition of MHORPO. Below we provide an example of an obtained set of constraints. Solving these constraints requires on the one hand, as in the first-order case, to automatically generate the adequate quasi-orderings $\succeq_I$ and $\succeq_Q$ (note that we are using the same kind of interpretations as in the first-order case) and on the other hand to adapt the notion of dependency graph for the higher-order case.

In the remaining of this section we give, by means of an example, the ideas for extracting a constraint based termination proof method from MHOSPO. We consider again the example 1. Applying the definition of MHOSPO and HOSPO on the rules, we can extract a constraint on $\succeq_I$ and a disjunction of constraints on $\succeq_Q$, depending on the cases we follow. In particular, we will present here the constraint obtained by applying cases 2 and 4 when the left-hand side term in the comparison is headed by a function symbol, and using case 1 (to conclude) when we have a subterm that can be reached (through equivalent types). Finally we use the definition of the computable closure when we have to check recursively the ordering on a term with a non-equivalent type. In this case we stop applying the definition of the closure when we arrive at a bound variable or at reachable subterm (through equivalent types) of an argument of the left-hand side term. With this strategy we will get the following constraints, denoting by $\mathcal{I}$ the constraint on $\succeq_I$ and by $\mathcal{Q}$ the constraint on $\succeq_Q$.

$$
\begin{aligned}
\mathcal{I}:\ & map(F,[]) \succeq_I [] \wedge \\
& map(F, cons(x, xs)) \succeq_I cons(@(F,x), map(F, xs)) \wedge \\
& ps([]) \succeq_I [] \wedge \\
& ps(cons(x, xs)) \succeq_I cons(x, ps(map(\lambda y.x + y, xs)))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{Q}:\ & map(F, cons(x, xs)) \succ_Q cons(@(F,x), map(F, xs)) \wedge \\
& map(F, cons(x, xs)) \succ_Q map(F, xs) \wedge \\
& ps(cons(x, xs)) \succ_Q cons(x, ps(map(\lambda y.x + y, xs))) \wedge \\
& ps(cons(x, xs)) \succ_Q ps(map(\lambda y.x + y, xs)) \wedge \\
& ps(cons(x, xs)) \succ_Q map(\lambda y.x + y, xs)\ \wedge\ ps(cons(x, xs)) \succ_Q x + y
\end{aligned}
$$

By a very simple cycle analysis on the constraint $\mathcal{Q}$ we can see that we have to define $\succeq_Q$ with a lexicographic combination with first component a precedence $\succeq_{\mathcal{P}}$ defined by $ps \succ_{\mathcal{P}} map \succ_{\mathcal{P}} cons$ and $ps \succ_{\mathcal{P}} +$. To decide the other component(s) of $\succeq_Q$ we can see how remains $\mathcal{Q}$ after the simplification using the precedence (we will keep using $\succeq_Q$ for the rest of components).

$$
\begin{aligned}
\mathcal{Q}:\ & map(F, cons(x, xs)) \succ_Q map(F, xs) \wedge \\
& ps(cons(x, xs)) \succ_Q ps(map(\lambda y.x + y, xs))
\end{aligned}
$$

Now we may rename the head symbols as done in the dependency pair method (using e.g. capital letters), and apply $\succeq_I$ afterwards. Then the resulting constraint on $\succeq_I$ should be the following (we keep using $\mathcal{Q}$ to denote the part of the constraint coming from $\succeq_Q$):

$$\begin{aligned}\mathcal{I}: &\ map(F,[]) \succeq_I [] \ \wedge\ map(F, cons(x,xs)) \succeq_I cons(@(F,x), map(F,xs)) \wedge \\ &\ ps([]) \succeq_I [] \ \wedge\ ps(cons(x,xs)) \succeq_I cons(x, ps(map(\lambda y.x+y, xs)))\end{aligned}$$

$$\begin{aligned}\mathcal{Q}: &\ MAP(F, cons(x,xs)) \succ_I MAP(F,xs) \wedge \\ &\ PS(cons(x,xs)) \succ_I PS(map(\lambda y.x+y, xs))\end{aligned}$$

Note that the resulting constraint look very similar to the output constraints of the dependency pair method in the first-order case. Thus, we can use similar techniques to guess the appropriate interpretation satisfying the constraint.

9 Conclusions

In this paper we have presented a new ordering-based method for proving termination of higher-order rewriting. The method properly extends both MSPO (in the first-order case) and HORPO (in the higher-order case). The method can be automated and its power has been shown by means of several examples which could not be handled by the previous methods.

Finally let us mention some work already in progress and some future work we plan to do.

1. We can adapt, in the same way as it can be done for HORPO [JR01], the method to be applicable to higher-order rewriting "à la Nipkow" [MN98], i.e. rewriting on terms in η-long β-normal form.
2. We will study other possible interpretations to build $\succeq_I$ using functionals in a similar way as in [Pol96], but with two relevant differences. First due to the fact that we are building a quasi-ordering we can use *weakly monotonic functionals* instead of *strict functionals*. Second, since we are not working on terms in η-long β-normal forms, we have to study whether we need to define $\succeq_I$ by first obtaining η-long β-normal form and then interpreting the normalized terms by using functionals. Note that if we normalize first, $\succeq_I$ will include trivially β-reduction.
3. We want to add to HOSPO more powerful cases to deal with terms headed by lambdas, which is, by now, the main weakness of the ordering, as well as some other improvements that have already been added to the initial version of HORPO [JR01].
4. We want to analyze the relationship between our method and a recent constraint-based method developed in [Pie01] for proving termination of higher-order logic programs.

References

[AG00] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[Bar92] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay and T. S. E. Maibaum, eds., *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

[BFR00] C. Borralleras, M. Ferreira, and A. Rubio. Complete monotonic semantic path orderings. *Proc. of the 17th International Conference on Automated Deduction*, *LNAI* 1831, pp. 346–364, Pittsburgh, USA, 2000. Springer-Verlag.

[Der82] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In Jan van Leeuwen, ed., *Handbook of Theoretical Computer Science*, vol. B: Formal Models and Semantics, chap. 6, pp. 244–320. Elsevier Science Publishers B.V., 1990.

[JR98] J-P. Jouannaud and A. Rubio. Rewite orderings for higher-order terms in η-long β-normal form and the recursive path ordering. *Theoretical Computer Science*, 208:33–58, 1998.

[JR99] J-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *14th IEEE Symposium on Logic in Computer Science (LICS)*, pp. 402–411, 1999.

[JR01] J.-P. Jouannaud and A. Rubio. Higher-order Recursive Path Orderings à la carte (draft), 2001.

[KL80] S. Kamin and J.-J. Lévy. Two generalizations of the recursive path ordering. Unpublished note, Dept. of Computer Science, Univ. of Illinois, Urbana, IL, 1980.

[LSS92] C. Loría-Sáenz and J. Steinbach. Termination of combined (rewrite and λ-calculus) systems. In *Proc. 3rd Int. Workshop on Conditional Term Rewriting Systems*, LNCS 656, pp. 143–147, 1992. Springer-Verlag.

[MN98] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192(1):3–29, 1998.

[Pie01] Brigitte Pientka. Termination and reduction checking for higher-order logic programs. *Proc. First International Joint Conference on Automated Reasoning*, volume 2083 of *LNAI*, pp. 402–415, 2001. Springer-Verlag.

[Pol96] J. van de Pol. *Termination of Higher-Order Rewrite Systems*. PhD thesis, Departament of Philosophy–Utrecht University, 1996.

[PS95] J. van de Pol and H. Schwichtenberg. Strict functional for termination proofs. In *Proc. of the International Conference on Typed Lambda Calculi and Applications*, 1995.

[vR01] Femke van Raamsdonk. On termination of higher-order rewriting. *12th Int. Conf. on Rewriting Techniques and Applications*, LNCS 2051, pp. 261–275, 2001.