Polymorphic Higher-Order Recursive Path Orderings*

Jean-Pierre Jouannaud†
École Polytechnique
LIX, UMR CNRS 7646
91400 Palaiseau, FRANCE

Albert Rubio
Technical University of Catalonia
Dep. LSI, Jordi Girona 1-3
08034 Barcelona, SPAIN

Abstract

This paper extends the termination proof techniques based on reduction orderings to a higher-order setting, by defining a family of recursive path orderings for terms of a typed lambda-calculus generated by a signature of polymorphic higher-order function symbols. These relations can be generated from two given well-founded orderings, on the function symbols and on the type constructors. The obtained orderings on terms are well-founded, monotonic, stable under substitution and include β-reductions. They can be used to prove the strong normalization property of higher-order calculi in which constants can be defined by higher-order rewrite rules using first-order pattern matching. For example, the polymorphic version of Gödel's recursor for the natural numbers is easily oriented. And indeed, our ordering is polymorphic, in the sense that a single comparison allows to prove the termination property of all monomorphic instances of a polymorphic rewrite rule. Many non-trivial examples are given which exemplify the expressive power of these orderings. All have been checked by our implementation.

This paper is an extended and improved version of [32]. Polymorphic algebras have been made more expressive than in our previous framework. The intuitive notion of a polymorphic higher-order ordering has now been made precise. The higher-order recursive path ordering itself has been made much more powerful by replacing the congruence on types used there by an ordering on types satisfying some abstract properties. Besides, using a restriction of Dershowitz's recursive path ordering for comparing types, we can integrate both orderings into a single one operating uniformly on both terms and types.

1 Introduction

Rewrite rules are used in programming languages and logical systems, with three main goals: defining functions by pattern matching; encoding rule-based decision procedures; describing computations over lambda-terms used as a suitable abstract syntax for encoding functional objects like programs or specifications. ML and the current version of Coq based on the Calculus of Inductive Constructions [14] exemplify the first use. A prototype version of Coq exemplifies the second use [9]. Alf [15] and Isabelle [44] exemplify the third use. In Isabelle, rules operate on terms in β-normal, η-expanded form and use higher-order pattern matching. In ML and Coq, they operate on arbitrary terms and use first-order pattern-matching. Both kinds of rules target different needs, and should, of course, coexist.

*This work was partly supported by the RNTL project AVERROES, France-Telecom, and the CICYT project LOGICTOOLS, ref. TIN2004-07925.
†Project LogiCal, Pôle Commun de Recherche en Informatique du Plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

The use of rules in logical systems is subject to three main meta-theoretic properties : type preservation, local confluence, and strong normalization. The first two are usually easy. The last one is difficult, requiring the use of sophisticated proof techniques based, for example, on Tait and Girard's computability predicate technique [22]. Our ambition is to provide a remedy for this situation by developing for the higher-order case the kind of automatable termination proof techniques that are available for the first-order case, of which the most popular one is the recursive path ordering [16].

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

In a preliminary version of this work presented at the Federated Logic Conference in Trento, our ordering could only compare terms of equal types (after identifying sorts such as Nat or List). In the present version, the ordering is capable of ordering terms of *decreasing types*, the ordering on types being simply a slightly weakened form of Dershowitz's recursive path ordering. Several other improvements have been made, which allow to prove a great variety of practical examples. To hint at the strength of our ordering, let us mention that the polymorphic version of Gödel's recursor for the natural numbers is easily oriented. And indeed, our ordering can prove at once the termination property of all monomorphic instances of a polymorphic rewrite rule. Many other examples are given which exemplify the expressive power of the ordering.

In the literature, one can find several attempts at designing methods for proving strong normalization of higher-order rewrite rules based on ordering comparisons. These orderings are either quite weak [38, 31], or need a heavy user interaction [47]. Besides, they operate on terms in η-long β-normal form, hence apply only to the higher-order rewriting "à la Nipkow" [40], based on higher-order pattern matching modulo $\beta\eta$. To our knowledge, our ordering is the first to operate on arbitrary higher-order terms, therefore applying to the other kind of rewriting, based on plain pattern matching. It is also the first to be automatable: an implementation is provided which does not require user-interaction. And indeed we want to stress several important features of our approach. Firstly, it can be seen as a way to lift an ordinal notation operating on a first-order language (here, the set of labelled trees ordered by the recursive path ordering) to an ordinal notation of higher type operating on a set of well-typed λ-expressions built over the first-order language. Secondly, the analysis of our ordering, based on Tait and Girard's computability predicate proof technique, leads to hiding this technique away, by allowing one to carry out future meta-theoretical investigations based on ordering comparisons rather than by a direct use of the computability predicate technique. Thirdly, a very elegant presentation of the whole ordering machinery obtained by integrating both orderings on terms and types into a single one operating on both kinds shows that this presentation can in turn be the basis for generalizing the ordering to dependent type calculi. Last, a modification of our ordering described in a companion paper can be used to prove strong normalization of higher-order rewrite rules operating on terms in η-long β-normal form and using higher-order pattern matching for firing rules [33].

Described in Section 2, our framework for rewriting includes several novel aspects, among which two important notions of polymorphic higher-order rewriting and of polymorphic higher-order rewrite order-

ings. In order to define these notions precisely, we first introduce polymorphic higher-order algebras. Types will therefore play a central role in this paper, but the reader should be aware that the paper is by no means about polymorphic typing : it is about polymorphic higher-order orderings. In particular, we could have also considered inductive types, without having to face unpredictable difficulties. We chose not to do so in the present framework, which, we think, is already quite powerful and complex, to a point that those readers who feel no particular interest in typing technicalities should probably restrict their first reading of this section to the type system of Figure 1, Definition 2.17 and Theorem 2.18. The rest of the paper, we think, can be understood without knowing the details of the typing apparatus. The basic version of our ordering is defined and studied in Section 3, where several examples are given. The notion of computability closure [6] used to boost the expressiveness of the ordering is introduced and studied in Section 4. Our prototype implementation is discussed in Section 5. Related work and potential extensions are discussed in Section 6.

The reader is expected to have some familiarity with the basics of term rewriting systems [17, 37, 1, 4] and typed lambda calculi [2, 3].

2 Polymorphic Higher-Order Algebras

This section describes polymorphic higher-order algebras, a higher-order algebraic framework with typing "à la ML" which makes it possible to precisely define what are polymorphic higher-order rewriting and polymorphic higher-order rewrite orderings. Our target is indeed to have higher-order rewrite rules in the calculus of constructions and check their termination property by means of orderings, but, although our results address polymorphic typing, they do not scale up yet to dependent types.

We define our typing discipline in Sections 2.1 to 2.5, before investigating their properties in Section 2.6, and then define higher-order rewriting in Section 2.8 and finally polymorphic higher-order rewrite orderings in Section 2.9. We conclude in Section 2.10 that polymorphic higher-order rewrite orderings allow showing that the derivation relation defined by a set of polymorphic rewrite rules is well-founded by checking the rules for decreasingness.

2.1 Types

Given a set $\mathcal{S}$ of *sort symbols* of a fixed arity, denoted by $s : *^n \Rightarrow *$, and a denumerable set $\mathcal{S}^\forall$ of *type variables*, the set $\mathcal{T}_{\mathcal{S}^\forall}$ of (first-order) *types* is generated by the following grammar:

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

We denote by $\mathcal{V}ar(\sigma)$ the set of type variables of the type $\sigma \in \mathcal{T}_{\mathcal{S}^\forall}$, and by $\mathcal{T}_\mathcal{S}$ the set of *monomorphic* or *ground* types, whose set of type variables is empty. Types in $\mathcal{T}_{\mathcal{S}^\forall} \setminus \mathcal{T}_\mathcal{S}$ are *polymorphic*.

Types are *functional* when headed by the $\to$ symbol, and *data types* when headed by a sort symbol. As usual, $\to$ associates to the right. We will often make explicit the functional structure of an arbitrary type τ by writing it in the *canonical form* $\sigma_1 \to \ldots \to \sigma_n \to \sigma$, with $n \geq 0$, where n is the *arity* of τ and σ is a data type or a type variable called *canonical output type* of τ. A *basic type* is either a data type or a type variable.

A *type substitution* is a mapping from $\mathcal{S}^\forall$ to $\mathcal{T}_{\mathcal{S}^\forall}$ extended to an endomorphism of $\mathcal{T}_{\mathcal{S}^\forall}$. We use postfix notation for their application to types or to other type substitutions. We denote by $\mathcal{D}om(\xi) = \{\alpha \in \mathcal{S}^\forall \mid \alpha\xi \neq \alpha\}$ the *domain* of ξ, and by $\mathcal{R}an(\xi) = \bigcup_{\alpha \in \mathcal{D}om(\xi)} \mathcal{V}ar(\alpha\xi)$ its *range*. Note that all variables in

$\mathcal{R}an(\xi)$ belong to $\mathcal{S}^\forall$ by assumption, that is, they must be declared beforehand. A type substitution σ is *ground* if $\mathcal{R}an(\sigma) = \emptyset$, and a *type renaming* if it is bijective.

We use α, β for type variables, $\sigma, \tau, \rho, \theta$ for arbitrary types and ξ for type substitutions.

A *unification problem* is a conjunction of equation among types such as $\sigma_1 = \tau_1 \wedge \ldots \sigma_n = \tau_n$. A *solution* is a ground type substitution ξ such that $\sigma_i\xi = \tau_i\xi$ for all $i \in [1..n]$. Solutions are ground instances of a unique (up to renaming of type variables in its range) type substitution called the *most general unifier* of the problem and denoted by $mgu(\sigma_1 = \tau_1 \wedge \ldots \sigma_n = \tau_n)$, a result due to Robinson.

2.2 Signatures

We are given a set of function symbols denoted by the letters f, g, h, which are meant to be algebraic operators equipped with a fixed number n of arguments (called the *arity*) of respective types $\sigma_1 \in \mathcal{T}_{\mathcal{S}^\forall}, \ldots, \sigma_n \in \mathcal{T}_{\mathcal{S}^\forall}$, and *output type* $\sigma \in \mathcal{T}_{\mathcal{S}^\forall}$. Let $\mathcal{F} = \biguplus_{\sigma_1,\ldots,\sigma_n,\sigma} \mathcal{F}_{\sigma_1\times\ldots\times\sigma_n\Rightarrow\sigma}$ be the set of all function symbols. The membership of a given function symbol f to a set $\mathcal{F}_{\sigma_1\times\ldots\times\sigma_n\Rightarrow\sigma}$ is called a *type declaration* and written $f : \sigma_1 \times \ldots \times \sigma_n \Rightarrow \sigma$. This type declarations is not a type, but corresponds to the type $\forall\mathcal{V}ar(\sigma_1)\ldots\forall\mathcal{V}ar(\sigma_n)\forall\mathcal{V}ar(\sigma)\sigma_1 \to \ldots \to \sigma_n \to \sigma$, hence, the intended meaning of a polymorphic type declaration is the set of all its monomorphic instances. In case $n = 0$, the declaration is written $f : \sigma$. A type declaration is *first-order* if its constituent types are solely built from sort symbols and variables, and higher-order otherwise. It is *polymorphic* if it uses some polymorphic type, otherwise, it is *monomorphic*. $\mathcal{F}$ is said to be *first-order* if all its type declarations are first-order, and *higher-order* otherwise. It is *polymorphic* if some type declaration is polymorphic, and *monomorphic* otherwise.

The triple $\mathcal{S}; \mathcal{S}^\forall; \mathcal{F}$ is called the *signature*. We sometimes say that the signature is *first-order, higher-order, polymorphic, monomorphic* when $\mathcal{F}$ satisfies the corresponding property.

2.3 Raw terms

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

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

Raw terms of the form $\lambda x : \sigma.u$ are called *abstractions*, while the other raw terms are said to be *neutral*. $@(u, v)$ denotes the application of u to v. We may sometimes omit the type σ in $\lambda x : \sigma.u$ as well as the application operator, writing $u(v)$ for $@(u, v)$, in particular when u is a higher-order variable. As a matter of convenience, we may write $u(v_1, \ldots, v_n)$, or $@(u, v_1, \ldots, v_n)$ for $u(v_1)\ldots(v_n)$, assuming $n \geq 1$. The raw term $@(u, v_1, \ldots, v_n)$ is called a (partial) *left-flattening* of $s = u(v_1)\ldots(v_n)$, u being possibly an application itself (hence the word "partial").

Note that our syntax requires using explicit applications for variables, since they cannot take arguments. On the other hand, function symbols have arities, eliminating the need for an explicit application of a function symbol to its arguments. Curried function symbol have arity zero, hence must be applied.

Raw terms are identified with finite labeled trees by considering $\lambda x : \sigma._$, for each variable x and type σ, as a unary function symbol taking a raw term u as argument to construct the raw term $\lambda x : \sigma.u$. We denote the set of free variables of the raw term t by $\mathcal{V}ar(t)$, its set of bound variables by $\mathcal{BV}ar(t)$, its size (the number of symbols occurring in t) by $|t|$. The notation $\overline{s}$ will be ambiguously used to denote a list, or a multiset, or a set of raw terms $s_1, \ldots, s_n$.

Positions are strings of positive integers. $\wedge$ and $\cdot$ denote respectively the empty string (root position) and the concatenation of strings. We use $\mathcal{P}os(t)$ for the set of positions in t. The *subterm* of t at position

p is denoted by $t|_p$, and we write $t \trianglerighteq t|_p$ for the subterm relationship. The result of replacing $t|_p$ at position p in t by u is denoted by $t[u]_p$. We sometimes use $t[x:\sigma]_p$ for a raw term with a (unique) hole of type σ at position p, also called a *context*.

Type substitutions are extended to terms as homomorphisms by letting $x\xi = x$, $@(u,v)\xi = @(u\xi, v\xi)$, $f(\bar{t})\xi = f(\bar{t}\xi)$ and $(\lambda x:\sigma.u)\xi = \lambda x:\sigma\xi.u\xi$.

2.4 Environments

Definition 2.1 *A* variable environment Γ *is a finite set of pairs written as* $\{x_1:\sigma_1,\ldots,x_n:\sigma_n\}$ *such that* $x_i \in \mathcal{X}$, $\sigma_i \in \mathcal{T}_{\mathcal{S}^\forall}$, *and* $x_i \neq x_j$ *for* $i \neq j$. $\mathcal{V}ar(\Gamma) = \{x_1,\ldots,x_n\}$ *is the set of variables of* Γ. *Given two variable environments* Γ *and* Γ', *their* composition *is the variable environment* $\Gamma \cdot \Gamma' = \Gamma' \cup \{x : \sigma \in \Gamma \mid x \notin \mathcal{V}ar(\Gamma')\}$. *Two variable environments* Γ *and* Γ' *are* compatible *if* $\Gamma \cdot \Gamma' = \Gamma \cup \Gamma'$.

We now collect all declarations into a single *environment* $\Sigma;\Gamma$, where the signature $\Sigma = \mathcal{S};\mathcal{S}^\forall;\mathcal{F}$ is the fixed part of the environment. We assume that there is exactly one declaration for each symbol in an environment $\Sigma;\Gamma$.

Example 1 We give here the signature for the specification of Gödel's system T. In contrast with Gödel's formulation, we use polymorphism to have the recursor rule as a polymorphic rewrite rule instead of a rule schema.

$\Sigma = \{\mathbb{N}:*;\ \alpha:*;\ 0:\mathbb{N},\ s:\mathbb{N}\Rightarrow\mathbb{N},\ +:\mathbb{N}\times\mathbb{N}\Rightarrow\mathbb{N},\ rec:\mathbb{N}\times\alpha\times(\mathbb{N}\to\alpha\to\alpha)\Rightarrow\alpha\}$.

$\Gamma = \{x:\mathbb{N}, U:\alpha,\ X:\mathbb{N}\to\alpha\to\alpha\}$.

Gödel's recursor rules are given in Example 2. □

2.5 Typing Rules

Typing rules restrict the set of raw terms by constraining them to follow a precise discipline. Our *principal typing judgements* are written as $\Gamma \vdash^c_\Sigma s:\sigma$, and read "$s$ has principal type σ in the environment Γ". The typing judgements are displayed in Figure 1.

Variables:

$$\frac{x:\sigma \in \Gamma}{\Gamma \vdash^c_\Sigma x:\sigma}$$

Abstraction:

$$\frac{\Gamma\cdot\{x:\sigma\} \vdash^c_\Sigma t:\tau}{\Gamma \vdash^c_\Sigma (\lambda x:\sigma.t):\sigma\to\tau}$$

Application:

$$\frac{\Gamma \vdash^c_\Sigma s:\sigma \quad \Gamma \vdash^c_\Sigma t:\tau \quad \xi \text{ most general unifier of } \alpha\to\beta = \sigma\wedge\alpha = \tau}{\Gamma \vdash^c_\Sigma @(s,t):\beta\xi}$$

Functions:

$$\frac{f:\sigma_1\times\ldots\times\sigma_n\Rightarrow\sigma\in\mathcal{F} \quad \Gamma\vdash^c_\Sigma t_1:\tau_1\ \ldots\ \Gamma\vdash^c_\Sigma t_n:\tau_n \quad \xi \text{ most general unifier of } \sigma_1=\tau_1\wedge\ldots\wedge\sigma_n=\tau_n}{\Gamma\vdash^c_\Sigma f(t_1,\ldots,t_n):\sigma\xi}$$

Figure 1. Typing judgements in higher-order algebras

According to the intended meaning of type declarations, we assume that the declaration $f:\sigma_1\times\ldots\times\sigma_n\Rightarrow\sigma$ is renamed so as to ensure that $\mathcal{V}ar(\sigma_1,\ldots,\sigma_n,\sigma)\cap\mathcal{V}ar(\tau_1,\ldots,\tau_n,\tau)=\emptyset$. Further, since the last two rules introduce a type substitution ξ, we shall consider for uniformity reason that the first two introduce the identity type substitution. Note also that the rule for applications is nothing but the Rule **Functions** applied to the symbol $@:(\alpha\to\beta)\times\alpha\Rightarrow\beta$.

Example 2 [Example 1 continued] Let us type check in the environment $\Sigma;\{\}$ the ground raw term $rec(s(0),0,rec(0,\lambda\, x:\mathbb{N}\, y:\mathbb{N}.+(x,y),\lambda\, x:\mathbb{N}\, y:\mathbb{N}\to\mathbb{N}\to\mathbb{N}\, z:\mathbb{N}.y(+(x,z))))$:

$$\dfrac{\dfrac{\dfrac{\overline{\{\}\ \vdash^c_\Sigma 0:\mathbb{N}}}{\{\}\ \vdash^c_\Sigma s(0):\mathbb{N}}\quad\dfrac{}{\{\}\ \vdash^c_\Sigma 0:\mathbb{N}}\quad\dfrac{\text{The three required premises are proved below}}{\{\}\ \vdash^c_\Sigma rec(0,\lambda x,y:\mathbb{N}.+(x,y),\lambda x:\mathbb{N}\, y:U\, z:\mathbb{N}.y(+(x,z))):\mathbb{N}}}{\{\}\ \vdash^c_\Sigma rec(s(0),0,rec(0,\lambda\, x:\mathbb{N}\, y:\mathbb{N}.+(x,y),\lambda\, x:\mathbb{N}\, y:\mathbb{N}\to\mathbb{N}\to\mathbb{N}\, z:\mathbb{N}.y(+(x,z)))):\mathbb{N}}$$

in which U is an abbreviation for the type $\mathbb{N}\to\mathbb{N}\to\mathbb{N}$. We start with the first two required premises:

$$\dfrac{}{\{\}\ \vdash^c_\Sigma 0:\mathbb{N}}\qquad\dfrac{\dfrac{\overline{\{x,y:\mathbb{N}\}\ \vdash^c_\Sigma x:\mathbb{N}}\quad\overline{\{x,y:\mathbb{N}\}\ \vdash^c_\Sigma y:\mathbb{N}}}{\{x,y:\mathbb{N}\}\ \vdash^c_\Sigma\ +(x,y):\mathbb{N}}}{\{\}\ \vdash^c_\Sigma \lambda x,y:\mathbb{N}.+(x,y):\mathbb{N}}$$

ending up with the third, in which Θ is an abbreviation for the environment $\{x,z:\mathbb{N},\ y:\mathbb{N}\to\mathbb{N}\to\mathbb{N}\}$:

$$\dfrac{\dfrac{\dfrac{}{\Theta\ \vdash^c_\Sigma y:\mathbb{N}\to(\mathbb{N}\to\mathbb{N})}\quad\dfrac{\overline{\Theta\ \vdash^c_\Sigma x:\mathbb{N}}\quad\overline{\Theta\ \vdash^c_\Sigma z:\mathbb{N}}}{\Theta\ \vdash^c_\Sigma\ +(x,z):\mathbb{N}}}{\{x,z:\mathbb{N}\ y:\mathbb{N}\to\mathbb{N}\to\mathbb{N}\}\ \vdash^c_\Sigma y(+(x,z)):\mathbb{N}\to\mathbb{N}}}{\{\}\ \vdash^c_\Sigma \lambda x:\mathbb{N}\, y:\mathbb{N}\to\mathbb{N}\to\mathbb{N}\, z:\mathbb{N}\,.\, y(+(x,z)):\mathbb{N}\to(\mathbb{N}\to\mathbb{N}\to\mathbb{N})\to\mathbb{N}\to\mathbb{N}\to\mathbb{N}}$$

Classically, we consider the proof of a given judgement $\Gamma\ \vdash^c_\Sigma s:\sigma$ as a tree whose nodes are labelled by the judgements derived in the proof in the following way: assuming that $\Delta\ \vdash^c_\Sigma t:\tau$ labels the node at position p, and $\Delta'\ \vdash^c_\Sigma t_1:\tau_1,\ldots,\Delta'\ \vdash^c_\Sigma t_n:\tau_n$ are the premisses of the rule used to derive the judgement $\Delta\ \vdash^c_\Sigma t:\tau$, then the judgement $\Delta'\ \vdash^c_\Sigma t_i:\tau_i$ labels the i-th son of the node p, that is the node at position $p\cdot i$ ($i\in[1..n]$), and the substitution ξ introduced by the rule labels all edges going from the node p to its sons. For convenience, we will assume an incoming edge at the root of the tree labelled by the identity type substitution. We use as usual $\mathcal{P}os(P)$ for the set of positions of the proof tree P, and draw as usual the proof trees upside down (that is, like a biological tree).

Definition 2.2 *Given an environment $\Sigma;\Gamma$, a raw term s is* typable *with* principal type σ *if the judgement $\Gamma\ \vdash^c_\Sigma s:\sigma$ is provable in our system.*

Given an environment $\Sigma;\Gamma$, a raw term s is typable *with* type τ*, written $\Gamma\ \vdash_\Sigma s:\tau$, if $\Gamma\ \vdash^c_\Sigma s:\tau$ and $\tau=\sigma\xi$ for some type subtitution ξ.*

We define the proof tree of the *ordinary typing judgement* $\Gamma\ \vdash_\Sigma s:\tau$ as being a proof tree for the judgement $\Gamma\ \vdash^c_\Sigma s:\sigma$, in which the incoming edge at the root is now labelled by the substitution ξ satisfying $\tau=\sigma\xi$. The judgement $\Gamma\ \vdash^c_\Sigma s:\sigma$ appears then as a particular case of the judgement $\Gamma\ \vdash_\Sigma s:\tau$ when ξ is the identity.

2.6 Typing properties

We now come to some simple properties of our type system which are instrumental to develop a theory of polymorphic higher-order rewriting. All these properties are fairly standard and easily proved by induction on the type derivation for most of them. We therefore skip their proofs.

Lemma 2.3 *Given an environment $\Sigma;\Gamma$ and a raw term s, whether s is typable is decidable in linear time in the size of s. When s is typable, its typing derivation $\Gamma\ \vdash^c_\Sigma s:\sigma$ and principal type σ are unique (up to renaming of type variables), hence all its types τ are type instances of σ.*

Lemma 2.4 (weakening) *Assume given an environment $\Sigma;\Gamma$, a term s and a type σ such that $\Gamma \vdash_\Sigma s : \sigma$ holds. Then, $\Gamma \cdot \Gamma' \vdash_\Sigma s : \sigma$ for all Γ' compatible with Γ. Further, $\Gamma \cdot \Gamma' \vdash^c_\Sigma s : \sigma$ if $\Gamma \vdash_\Sigma s : \sigma$.*

Lemma 2.5 (strengthening) *Assume given an environment $\Sigma;\Gamma \cdot \{x : \tau\} \cdot \Gamma'$, a term s such that $x \notin \mathcal{V}ar(s)$ and a type σ such that $\Gamma \cdot \{x : \tau\} \cdot \Gamma' \vdash_\Sigma s : \sigma$. Then, $\Gamma \cdot \Gamma' \vdash_\Sigma s : \sigma$. Further, $\Gamma \cdot \Gamma' \vdash^c_\Sigma s : \sigma$ if $\Gamma \cdot \{x : \tau\} \cdot \Gamma' \vdash^c_\Sigma s : \sigma$.*

Lemma 2.6 (subterm) *Let P denote the proof of the judgement $\Gamma \vdash^c_\Sigma s : \sigma$ and p be a position in $\mathcal{P}os(s)$. Then, there exists a unique position $q(p) \in \mathcal{P}os(P)$ such that the subproof $P|_{q(p)}$ is a proof of a judgement of the form $\Gamma_{s|_p} \vdash^c_\Sigma s|_p : \tau$.*

Definition 2.7 *Given a judgement $\Gamma \vdash_\Sigma s : \sigma$ and a position $p \in \mathcal{P}os(s)$, we call* actual type *of $s|_p$, the type $\theta = \tau\xi$ instance of the principal type τ of $s|_p$ in the environment $\Gamma_{s|_p}$ by the type substitution ξ labelling the arc ending at position $q(p)$.*

Example 3 [Example 1 continued] Let us type check in the environment $\Sigma;\{\}$ the simple raw term $s = rec(0, 0, \lambda\, x : \mathbb{N}\, y : \alpha.y)$ with the rules of Figure 1, writing the type substitution used in **Applications** and **Function** to the right of the rule when it is not the identity:

$$\dfrac{\dfrac{}{\{\} \vdash^c_\Sigma 0 : \mathbb{N}} \quad \dfrac{}{\{\} \vdash^c_\Sigma 0 : \mathbb{N}} \quad \dfrac{\dfrac{\dfrac{}{\{x : \mathbb{N}, y : \alpha\} \vdash^c_\Sigma .y : \alpha}}{\{x : \mathbb{N}\} \vdash^c_\Sigma \lambda\, y : \alpha.y : \alpha \to \alpha}}{\{\} \vdash^c_\Sigma \lambda\, x : \mathbb{N}\, y : \alpha.y : \mathbb{N} \to \alpha \to \alpha}}{\{\} \vdash^c_\Sigma rec(0, 0, \lambda\, x : \mathbb{N}\, y : \alpha.y : \mathbb{N} \to \alpha \to \alpha) : \mathbb{N}}\ \{\alpha \mapsto \mathbb{N}\}$$

The principal and actual types of all subterms of s coincide, except for its subterm $s|_3 = \lambda\, x : \mathbb{N}\, y : \alpha.y$ whose principal and actual type in the judgement $\{\} \vdash^c_\Sigma s : \mathbb{N}$ are respectively $\mathbb{N} \to \alpha \to \alpha$ and $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$. The latter is obtained from the former by the type instantiation $\{\alpha \mapsto \mathbb{N}\}$.

Given a typing judgement $\Gamma \vdash_\Sigma s : \sigma$ and a position $p \in \mathcal{P}os(s)$, the actual type θ of $s|_p$ plays a key role throughout the paper, and in the implementation as well.

Lemma 2.8 (replacement) *Assume given an environment $\Sigma;\Gamma$, two terms s and v, two types σ and τ, and a position $p \in \mathcal{P}os(s)$ such that $\Gamma \vdash_\Sigma s : \sigma$, $\Gamma_{s|_p} \vdash_\Sigma s|_p : \tau$ where τ is the actual type of $s|_p$, and $\Gamma_{s|_p} \vdash_\Sigma v : \tau$. Then, $\Gamma \vdash_\Sigma s[v]_p : \sigma$.*

As a consequence of the replacement lemma, we will use ordinary judgements only in the sequel.

2.7 Substitutions

Definition 2.9 *A* term substitution, *or simply* substitution *is a finite set $\gamma = \{(x_1 : \sigma_1) \mapsto (\Gamma_1, t_1); \ldots; (x_n : \sigma_n) \mapsto (\Gamma_n, t_n)\}$, whose elements are quadruples made of a variable symbol, a type, a term environment and a term, such that*

(i) for each $i \in [1..n]$, t_i is typable in Γ_i with principal type τ_i;
(ii) $\forall i \neq j \in [1..n]$, Γ_i and Γ_j are compatible environments;
(iii) the type unification problem $\sigma_1 = \tau_1 \wedge \ldots \wedge \sigma_n = \tau_n$ has a most general unifier ξ_γ;
(iv) $\forall i \in [1..n]$, $t_i \neq x_i$ or $\sigma_i\xi_\gamma \neq \sigma_i$ and $\forall i \neq j \in [1..n]$, $x_i \neq x_j$;

The substitution γ is type preserving *if ξ_γ is the identity, and a* specialization *if $t_i = x_i$ for all i.*

The domain of the substitution γ is the environment $\mathcal{D}om(\gamma) = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ while its range *is the environment $\mathcal{R}an(\gamma) = \bigcup_{i\in[1..n]} \Gamma_i$. We say that $x \in \mathcal{D}om(\gamma)$ whenever $x : \sigma \in \mathcal{D}om(\Gamma)$ for some type σ. We sometimes omit the parentheses, the type σ_i and the environment Γ_i in $(x_i : \sigma_i) \mapsto (\Gamma_i, t_i)$.*

Note that the set $\mathcal{R}an(\gamma)$ is indeed an environment by our compatibility assumption (iv), which is of course compatible with every environment Γ_i.

The need of a term environment Γ_i for each variable x_i comes from the requirement of typing the term t_i (with a type compatible with the type σ_i of the variable x_i that it will be substituted to, a property ensured by condition (iii)). In case t_i is ground, its typing environment is of course empty. This is the case in the coming example:

Example 4 [Example 3 continued] Here is an example of a ground substitution, for which condition (i) have been proved in Example 3. The other conditions are easily checked here (condition (iii) is satisfied with the substitution $\xi_\gamma = \alpha \mapsto \mathbb{N}$):

$$\begin{array}{rcl} \{(x : \mathbb{N}) & \mapsto & (\{\}, s(0)); \\ (U : \mathbb{N}) & \mapsto & (\{\}, 0); \\ (X : \mathbb{N} \to \mathbb{N} \to \mathbb{N}) & \mapsto & (\{\}, \lambda\, x : \mathbb{N}\, y : \alpha.y)\} \end{array}$$

In general, condition (iii) may yield a non-trivial unification problem as in the following example:

Example 5 Let
$\Sigma = \{\mathbb{N}, \mathbb{B} : *, pair : *^2 \Rightarrow *, list : * \Rightarrow *\}; \{\alpha, \beta, \alpha' : *\}; \{iter : (\alpha \to \beta) \times list(\alpha) \Rightarrow list(\beta)\}$.

Consider the term $iter(F, l)$ in the environment $\{F : \alpha \to \beta, l : list(\alpha)\}$ which will be used in the rest of the example as the domain for a substitution γ. We have the typing derivation:
$\{F : \alpha \to \beta, l : list(\alpha)\} \vdash_\Sigma iter(F, l) : list(\beta)$

Let us now first consider a non-substitution fullfilling all conditions but (iii):
$\gamma = \{(F : \alpha \to \beta) \mapsto (\{F : pair(\mathbb{N}, \alpha') \to \alpha'\}, F), (l : list(\alpha)) \mapsto (\{l : list(pair(\alpha', \mathbb{B}))\}, l)\}$.
Then, the unification problem originating from condition (iii), $\alpha \to \beta = pair(\mathbb{N}, \alpha') \to \alpha' \wedge list(\alpha) = list(pair(\alpha', \mathbb{B}))$ fails by generating the unsolvable equation $\mathbb{N} = \mathbb{B}$.
Trying to type the same term $iter(F, l)$ in the environment $\mathcal{R}an(\gamma) = \{F : pair(\mathbb{N}, \alpha') \to \alpha', l : list(pair(\alpha', \mathbb{B}))\}$ results in a failure, since the **Functions** rule generates the very same unsolvable unification problem as before.

Let us finally consider the substitution (indeed, a specialization)
$\gamma = \{(F : \alpha \to \beta) \mapsto (\{F : pair(\mathbb{N}, \alpha') \to \alpha'\}, F), (l : list(\alpha)) \mapsto (\{l : list(pair(\alpha', \mathbb{N}))\}, l)\}$.
We check condition (iii) by solving the unification problem $\alpha \to \beta = pair(\mathbb{N}, \alpha') \to \alpha' \wedge list(\alpha) = list(pair(\alpha', \mathbb{N}))$, which has most general unifier $\{\alpha \mapsto pair(\mathbb{N}, \mathbb{N}), \alpha' \mapsto \mathbb{N}, \beta \mapsto \mathbb{N}\}$.
Trying now to type the term $iter(F, l)$ in the environment $\mathcal{R}an(\gamma) = \{F : \alpha \to \beta, l : list(\alpha)\}$, we get:
$\{F : pair(\mathbb{N}, \alpha') \to \alpha', l : list(pair(\alpha', \mathbb{N}))\} \vdash_\Sigma iter(F, l) : list(\mathbb{N}))$
We observe that the obtained type for $iter(F, l)$ is the instance of its type in the environment $\mathcal{D}om(\gamma) = \{F : \alpha \to \beta, l : list(\alpha)\}$ by the type substitution ξ_γ. □

Definition 2.10 *A substitution γ is said to be* compatible *with an environment Γ if*
(i) $\mathcal{D}om(\gamma)$ is compatible with Γ,
(ii) $\mathcal{R}an(\gamma)$ is compatible with $\Gamma \setminus \mathcal{D}om(\gamma)$.
We will also say that γ is compatible with the judgement $\Gamma \vdash_\Sigma s : \sigma$.

Compatibility of a term substitution with an environment Γ is a necessary condition for defining how the substitution operates on a term typable in the environment Γ to yield a term typable in the environment $(\Gamma \setminus \mathcal{D}om(\gamma)) \cdot \mathcal{R}an(\gamma)$:

Definition 2.11 *A term substitution γ compatible with a judgement $\Gamma \vdash_\Sigma s : \sigma$ operates as an endomorphism on s (keeping its bound variables unchanged) and yields a term $s\gamma$ defined as follows:*

If	$s = x \in \mathcal{X}$ *and* $x \notin \mathcal{V}ar(\gamma)$	*then*	$s\gamma = x$
If	$s = x \in \mathcal{X}$ *and* $(x : \sigma) \mapsto (\Theta, t) \in \gamma$	*then*	$s\gamma = t$
If	$s = @(u, v)$	*then*	$s\gamma = @(u\gamma, v\gamma)$
If	$s = f(u_1, \ldots, u_n)$	*then*	$s\gamma = f(u_1\gamma, \ldots, u_n\gamma)$
If	$s = \lambda x : \tau.u$ *and* y *a fresh variable*	*then*	$s\gamma = \lambda y : \tau\xi_\gamma.(u\{(x : \tau) \mapsto (\{y : \tau\xi_\gamma\}, y)\})\gamma$

Term substitutions can be factored out via a specialization (possibly the identity) as follows:

Lemma 2.12 *Every term substitution γ can be written in the form $\delta\theta$ where δ is a specialization and θ is type preserving, that is, for any term s, $s\gamma = (s\delta)\theta$.*

Proof: Let $\gamma = \{(x_i : \sigma_i) \mapsto (\Gamma_i, t_i)\}_i$. Take $\delta = \{(x_i, \sigma_i) \mapsto (\{x_i : \sigma_i\xi_\gamma\}, x_i)\}$, and $\theta = \{(x_i, \sigma_i\xi_\gamma) \mapsto (\{\}, x_i\gamma)\}$. □

When writing $s\gamma$, using postfixed notation for substitutions, we will always make the assumption that the domain of γ is compatible with the judgement $\Gamma \vdash_\Sigma s : \sigma$. We will use the letter γ for arbitrary substitutions and the notation $A\gamma$, where A is a set of terms, for the set of instances of the terms in A.

Example 6 [Example 1 continued] Let us illustrate the use of term substitutions with the term $rec(s(0), 0, rec(0, \lambda\, x : \mathbb{N}\; y : \mathbb{N}. + (x, y), \lambda\, x : \mathbb{N}\; y : \mathbb{N} \to \mathbb{N} \to \mathbb{N}\; z : \mathbb{N}.y(+(x, z))))$, which happens to be an instance of the lefthand side of the second Gödel's recursor rule $rec(s(x), U, X) \to @(X, x, rec(x, U, X))$ for which $x : \mathbb{N}, U : \alpha$ and $X : \mathbb{N} \to \alpha \to \alpha$, by the substitution

$$\begin{array}{rcl} \{(x : \mathbb{N}) & \mapsto & (\{\}, s(0)) \\ (U : \alpha) & \mapsto & (\{\}, 0) \\ (X : \mathbb{N} \to \alpha \to \alpha) & \mapsto & (\{\}, rec(0, \lambda\, x : \mathbb{N}\; y : \mathbb{N}. + (x, y), \lambda\, x : \mathbb{N}\; y : \mathbb{N} \to \mathbb{N} \to \mathbb{N}\; z : \mathbb{N}.y(+(x, z))))\} \end{array}$$

while its subterm
$rec(0, \lambda\, x : \mathbb{N}\; y : \mathbb{N}. + (x, y), \lambda\, x : \mathbb{N}\; y : \mathbb{N} \to \mathbb{N} \to \mathbb{N}\; z : \mathbb{N}.y(+(x, z)))$ is an instance of the same lefthand side by the substitution

$$\begin{array}{rcl} \{(x : \mathbb{N}) & \mapsto & (\{\}, 0) \\ (U : \alpha) & \mapsto & (\{\}, \lambda\, x : \mathbb{N}\; y : \mathbb{N}. + (x, y)) \\ (X : \mathbb{N} \to \alpha \to \alpha) & \mapsto & (\{\}, \lambda\, x : \mathbb{N}\; y : \mathbb{N} \to \mathbb{N} \to \mathbb{N}\; z : \mathbb{N}.y(+(x, z)))\}. \end{array}$$

These substitutions are factored out by the respective specializations
$\{(U : \alpha) \mapsto (\{U : \mathbb{N}\}, U);\ (X : \mathbb{N} \to \alpha \to \alpha) \mapsto (\{X : \mathbb{N} \to \mathbb{N} \to \mathbb{N}\}, X)\}$ and
$\{(U : \alpha) \mapsto (\{U : \mathbb{N}\}, U);\ (X : \mathbb{N} \to \alpha \to \alpha) \mapsto (\{X : \mathbb{N} \to (\mathbb{N} \to \mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N} \to \mathbb{N}\}, X)\}$,
whose associated type instantiations are respectively $\alpha \mapsto \mathbb{N}$ and $\alpha \mapsto (\mathbb{N} \to \mathbb{N} \to \mathbb{N})$. □

The next lemma makes precise the action of substitutions on typing:

Lemma 2.13 *Assume given an environment $\Sigma; \Gamma$ and a substitution γ compatible with the judgement $\Gamma \vdash_\Sigma s : \sigma$. Then, $(\Gamma \setminus \mathcal{D}om(\gamma)) \cdot \mathcal{R}an(\gamma) \vdash_\Sigma s\gamma : \sigma\xi_\gamma$.*

Proof: The proof is done by induction on the derivation of the judgement $\Gamma \vdash_\Sigma s : \sigma$. We carry out the **Abstraction** case, which shows the need for instantiating the types of bound variables.

Assume that $s = \lambda x : \tau.u$ with $\sigma = \tau \to \sigma'$, hence $\Gamma \cdot \{x : \tau\} \vdash_\Sigma u : \sigma'$ by the rule **Abstraction**. Given now a fresh variable y, let us consider the substitution $\gamma' = \gamma \cup \{(x : \tau) \mapsto (\{y : \tau\xi_\gamma\}, y)\}$. Since Γ is compatible with γ and y is a fresh variable, Γ is compatible with γ'. By induction hypothesis, $(\Gamma \setminus \mathcal{D}om(\gamma')) \cdot \mathcal{R}an(\gamma') \vdash_\Sigma u\gamma' : \sigma\xi'_\gamma$. By construction, $\xi_{\gamma'} = \xi_\gamma$, and since y is a fresh variable,

$(\Gamma \setminus \mathcal{D}om(\gamma')) \cdot \mathcal{R}an(\gamma') = (\Gamma \setminus \mathcal{D}om(\gamma)) \cdot \mathcal{R}an(\gamma) \cdot \{y : \tau\xi_\gamma\}$. Therefore, our typing judgement becomes $(\Gamma \setminus \mathcal{D}om(\gamma)) \cdot \mathcal{R}an(\gamma) \cdot \{y : \tau\xi_\gamma\} \vdash_\Sigma u\{(x : \tau) \mapsto (\{y : \tau\xi_\gamma\}, y)\}\gamma : \sigma'\xi_\gamma$. By the rule **Abstraction**, we get $(\Gamma \setminus \mathcal{D}om(\gamma)) \cdot \mathcal{R}an(\gamma) \vdash_\Sigma \lambda y : \tau\xi_\gamma.u\{(x : \tau) \mapsto (\{y : \tau\xi_\gamma\}, y)\gamma : (\tau\xi_\gamma \to \sigma'\xi_\gamma) = \sigma\xi_\gamma$. Definition 2.11 now yields the expected result. □

Note that Lemma 2.5 allows us to clean the environment from the variables which do not occur in $s\gamma$.

As corollaries of Lemma 2.13, we obtain:

Corollary 2.13.1 *Let γ be a type preserving substitution compatible with the judgement $\Gamma \vdash_\Sigma s : \sigma$. Then, $(\Gamma \setminus \mathcal{D}om(\gamma)) \cdot \mathcal{R}an(\gamma) \vdash_\Sigma s\gamma : \sigma$.*

Corollary 2.13.2 *Let δ be a specialization compatible with the judgement $\Gamma \vdash_\Sigma s : \sigma$. Then, $s\delta = s\xi_\delta$ and $(\Gamma \setminus \mathcal{D}om(\delta)) \cdot \mathcal{R}an(\delta) \vdash_\Sigma s\delta : \sigma\xi_\delta$.*

2.8 Plain higher-order rewriting [28]

We now come to the definition of plain higher-order rewriting based on plain pattern-matching, as considered in Jouannaud and Okada [28] or in the Calculus of Inductive Constructions [14].

Definition 2.14 *Given a signature Σ, a* higher-order rewrite rule *or simply* rewrite rule *is a triple written $\Gamma \vdash l \to r$, where Γ is an environment and l, r are higher-order terms such that*

(i) $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \subseteq \mathcal{V}ar(\Gamma)$,

(ii) for all substitutions γ such that $\Gamma \vdash_\Sigma l\gamma : \sigma$, then $\Gamma \vdash_\Sigma r\gamma : \sigma$.

The rewrite rule is said to be polymorphic *if σ is a polymorphic type for some substitution γ.*

A plain higher order rewriting system, *or simply* rewriting system *is a set of higher-order rewrite rules.*

Example 7 Here is an example of a triple which is not a rewrite rule because it violates condition (ii): let $\mathcal{F} = \{f : \alpha \Rightarrow \alpha, g : \beta \Rightarrow \beta, 0 : \mathbb{N}\}$, $\mathcal{T}_{\mathcal{S}^\forall} = \{\alpha, \beta\}$, and consider the triple $\{x : \alpha\} \vdash f(x) \to g(0)$. We have $\{x : \alpha\} \vdash_\Sigma f(x) : \alpha$, $\{x : \alpha\} \vdash_\Sigma g(0) : \mathbb{N}$, and α has instances different from $\mathbb{N}$ such as $\mathbb{N} \to \mathbb{N}$. Therefore, the triple $\{x : \alpha\} \vdash f(x) \to g(0)$ is not a rule. On the other hand, the instance $\{x : \alpha\} \vdash f(0) \to g(0)$ is a rule. □

One may wonder whether a given triple $\Gamma \vdash l \to r$ is a rewrite rule, since condition (ii) quantifies over all possible substitutions. One can indeed answer the question, since types are first order terms and the existence of instances of a type σ (the principal type of s) which are not instances of τ (the principal type of r) can be expressed by the first-order formula $\exists\alpha.\alpha = \sigma \wedge \neg(\alpha = \tau)$, with α a fresh variable, interpreted over the set of ground types. Validity of such formulas is decidable by a result of Mal'cev [39], but it is clear that there is no α satisfying $\alpha = \sigma \wedge \neg(\alpha = \tau)$ iff σ is an instance of τ, that is, the principal type of the righthand side is more general that the principal type of the lefthand one. There are useful examples of such rules in practice.

We shall sometimes omit the environment Γ in the rule $\Gamma \vdash_\Sigma l \to r$ when it can be inferred from the context. The β- and η-rules below are two examples of polymorphic rewrite rule schemas:

$$\begin{array}{rcl} \{u : \alpha,\ v : \beta\} \vdash_\Sigma @(\lambda x : \alpha.v, u) & \longrightarrow_\beta & v\{x \mapsto u\} \\ \{x : \alpha,\ u : \alpha \to \beta\} \vdash_\Sigma \lambda x.@(u, x) & \longrightarrow_\eta & u \quad \text{if } x \notin \mathcal{V}ar(u) \end{array}$$

Making these rule schemas true rewrite rules is possible to the price of including variables with arities in our framework, an idea due to Klop [36].

We are now ready for defining the rewrite relation :

Definition 2.15 *Given a plain higher-order rewriting system R and an environment Γ, a term s such that $\Gamma \vdash_\Sigma s : \sigma$ rewrites to a term t at position p with the rule $\Theta \vdash l \to r$ and the term substitution γ, written $\Gamma \vdash s \xrightarrow[\Theta \vdash l \to r]{p} t$, or simply $\Gamma \vdash s \to_R t$, or even $s \to_R t$ assuming the environment Γ, if the following conditions are satisfied :*

(i) $\mathcal{D}om(\gamma) \subseteq \Theta$; *(ii)* $\Theta \cdot \mathcal{R}an(\gamma) \subseteq \Gamma_{s|_p}$; *(iii)* $s|_p = l\gamma$; *(iv)* $t = s[r\gamma]_p$.

Note that, by definition of higher-order substitutions, condition (iii) implies that variables which are bound in the rule do not occur free in γ, therefore avoiding capturing variables when rewriting.

Type checking the rewritten term is not necessary, thanks to the following property:

Lemma 2.16 (type preservation) *Assume that $\Gamma \vdash_\Sigma s : \sigma$ and $\Gamma \vdash s \to_R t$. Then $\Gamma \vdash_\Sigma t : \sigma$.*

Proof: Let $\Gamma \vdash s \xrightarrow[\Theta \vdash l \to r]{p} t$. By Lemma 2.6, $\Gamma_{s|_p} \vdash_\Sigma s|_p : \tau$, with τ the actual type of $s|_p$ in $\Gamma \vdash_\Sigma s : \sigma$. By condition (iii), $\Gamma_{s|_p} \vdash_\Sigma l\gamma : \tau$. By conditions (i) and (ii), the substitution γ is compatible with the environment Θ, and by condition (i) in Definition 2.14, $\mathcal{V}ar(l) \subseteq \mathcal{V}ar(\Theta) \subseteq \mathcal{V}ar(\Theta \cdot \mathcal{R}an(\gamma))$. By Lemma 2.5 (repeated) it follows that $\Theta \cdot \mathcal{R}an(\gamma) \vdash_\Sigma l\gamma : \tau$. By condition (ii) of Definition 2.14, $\Theta \cdot \mathcal{R}an(\gamma) \vdash_\Sigma r\gamma : \tau$. By conditions (ii) and Lemma 2.4, $\Gamma_{s|_p} \vdash_\Sigma r\gamma : \tau$. By Lemma 2.8, $\Gamma \vdash_\Sigma s[r\gamma]_p : \sigma$. We conclude with condition (iv). □

Example 8 [Example 1 continued] Here are Gödel's recursor rules for natural numbers:

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

Rewriting $rec(s(0), 0, rec(0, \lambda\, x : \mathbb{N}\ y : \mathbb{N}.\ +(x, y), \lambda\, x : \mathbb{N}\ y : \mathbb{N} \to \mathbb{N} \to \mathbb{N}\ z : \mathbb{N}.y(+(x, z))))$ with a call-by-value strategy (redexes are underlined) until a normal form is obtained, we get:

$$\begin{array}{ll} & rec(s(0), 0, \underline{rec(0, \lambda\, x : \mathbb{N}\ y : \mathbb{N}.\ +(x, y), \lambda\, x : \mathbb{N}\ y : \mathbb{N} \to \mathbb{N} \to \mathbb{N}\ z : \mathbb{N}.y(+(x, z)))}) \\ \to^3_{\{U:\alpha,\ X:\mathbb{N}\to\alpha\to\alpha\} \vdash rec(0,U,X)\to U} & \underline{rec(s(0), 0, \lambda\, x : \mathbb{N}\ y : \mathbb{N}.\ +(x, y))} \\ \to^\epsilon_{\{x:\mathbb{N},\ U:\alpha,\ X:\mathbb{N}\to\alpha\to\alpha\} \vdash rec(s(x),U,X)\to @(X,x,rec(x,U,X))} & \\ & \underline{@(\lambda\, x : \mathbb{N}\ y : \mathbb{N}.\ +(x, y), 0, rec(0, 0, \lambda\, x : \mathbb{N}\ y : \mathbb{N}.\ +(x, y)))} \\ \to^\epsilon_\beta & \underline{@(\lambda\, y : \mathbb{N}.\ +(0, y), rec(0, 0, \lambda\, x : \mathbb{N}\ y : \mathbb{N}.\ +(x, y)))} \\ \to^\epsilon_\beta & +(0, \underline{rec(0, 0, \lambda\, x : \mathbb{N}\ y : \mathbb{N}.\ +(x, y))}) \\ \to^2_{\{U:\alpha,\ X:\mathbb{N}\to\alpha\to\alpha\} \vdash rec(0,U,X)\to U} & \underline{+(0, 0)} \\ \to^\epsilon_{\{x:\mathbb{N}\} \vdash +(x,0)\to x} & 0 \end{array}$$

As a general benefit, the use of polymorphic signatures allows us to have only one recursor rule, instead of infinitely many rules described by one rule schema as in Gödel's original presentation. □

Other examples of higher-order rewrite systems are developed in section 3.

Because plain higher-order rewriting is type preserving, we may often omit the environment in which a term is type checked as well as its type, and consider the sequence of terms originating from a given term s type checked in an environment Γ by rewriting with a given set R of higher-order rules. In other words, we will often consider rewriting as a *relation on terms*, which will allow us to simplify our notations when needed.

A term s such that $s \xrightarrow[R]{p} t$ is called *reducible* (with respect to R). $s|_p$ is a *redex* in s, and t is the *reduct* of s. Irreducible terms are said to be in R*-normal form*. A substitution γ is in R-normal form if $x\gamma$ is

in R-normal form for all x. We denote by $\xrightarrow[R]{*}$ the reflexive, transitive closure of the rewrite relation $\xrightarrow[R]{}$, and by $\longleftrightarrow_R^*$ its reflexive, symmetric, transitive closure. We are indeed interested in the relation $\xrightarrow[R\beta\eta]{} = \longrightarrow_R \cup \longrightarrow_\beta \cup \longrightarrow_\eta$.

Given a rewrite relation $\longrightarrow$, a term s is *strongly normalizable* if there is no infinite sequence of rewrites issuing from s. A substitution γ is *strongly normalizable* if all terms $x\gamma$ such that $x \in \mathcal{D}om(\gamma)$ are strongly normalizable.

The rewrite relation itself is *strongly normalizing*, or *terminating*, if all terms are strongly normalizable. It is confluent if $s \longrightarrow^* u$ and $s \longrightarrow^* v$ implies that $u \longrightarrow^* t$ and $v \longrightarrow^* t$ for some t.

2.9 Higher-Order Reduction Orderings

We will make intensive use of well-founded relations for proving strong normalization properties, using the vocabulary of rewrite systems for these relations. For our purpose, these relations may not be transitive, hence are not necessarily orderings, although their transitive closures will be well-founded orderings, which justifies to sometimes call them orderings by abuse of terminology.

For our purpose, a *strict ordering* $>$ is an irreflexive transitive relation, an *ordering* $\geq$ is the union of its strict part $>$ with equality $=$, and a *quasi-ordering* $\succeq$ is the union of its strict part $\succ$ with its equivalence $\simeq$. Because we identify α-convertible higher-order terms, their equality contains implicitly α-conversion. The following results will play a key role, see [17]:

Assume $\succ, \succ_1, \ldots, \succ_n$ are relations on n given sets $S, S_1, \ldots, S_n$. Let

- $(\succ_1, \ldots, \succ_n)_{lex}$ be the relation on $\bigcup_{k \leq n} S_1 \times \ldots \times S_k$ defined as $(s_1, \ldots, s_m)(\succ_1, \ldots, \succ_n)_{lex}(t_1, \ldots, t_p)$ iff $\exists i \in [1..min(m,p)]$ such that $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$ and ($s_i \succ_i t_i$ or ($s_i = t_i$ and $i = p < m$);

- $\succ_{mul}$ be the relation on the set of multisets of elements of S defined as
$M \cup \{x\} \succ_{mul} N \cup \{y_1, \ldots, y_n\}$ iff $\forall i \in [1..n]\ x \succ y_i$ and $M = N$ or $M \succ_{mul} N$.

It is well known that both operations preserve the well-foundedness of the relations $\succ, \succ_1, \ldots, \succ_n$, and that they also preserve transitivity, hence orderings.

We end up this section by defining the notion of a reduction ordering operating on higher-order terms.

Definition 2.17 *A* higher-order reduction ordering $\succ$ *is a well-founded ordering of the set of judgements which is*

(i) monotonic*:* $(\Gamma \vdash_\Sigma s : \sigma) \succ (\Gamma \vdash_\Sigma t : \sigma)$ *implies that for all* Γ' *compatible with* Γ *such that* $\Gamma' \vdash_\Sigma u[x : \sigma] : \tau$*, then* $(\Gamma \cdot \Gamma' \vdash_\Sigma u[s] : \tau) \succ (\Gamma \cdot \Gamma' \vdash_\Sigma u[t] : \tau)$*;*

(ii) stable*:* $(\Gamma \vdash_\Sigma s : \sigma) \succ (\Gamma \vdash_\Sigma t : \sigma)$ *implies that for all type preserving substitution* γ *whose domain is compatible with* Γ*, then* $(\Gamma \cdot \mathcal{R}an(\gamma) \vdash_\Sigma s\gamma : \sigma) \succ (\Gamma \cdot \mathcal{R}an(\gamma) \vdash_\Sigma t\gamma : \sigma)$*;*

(iii) polymorphic*:* $(\Gamma \vdash_\Sigma s : \sigma) \succ (\Gamma \vdash_\Sigma t : \sigma)$ *implies that for all specialization* δ *whose domain is compatible with* Γ*, then* $(\Gamma \cdot \mathcal{R}an(\delta) \vdash_\Sigma s\delta : \sigma\xi_\delta) \succ (\Gamma \cdot \mathcal{R}an(\delta) \vdash_\Sigma t\gamma : \sigma\xi_\delta)$*;*

(iv) compatible*:* $(\Gamma \vdash_\Sigma s : \sigma) \succ (\Gamma \vdash_\Sigma t : \sigma)$ *implies* $(\Gamma' \vdash_\Sigma s : \sigma) \succ (\Gamma' \vdash_\Sigma t : \sigma)$ *for all environments* Γ' *such that* Γ *and* Γ' *are compatible,* $\Gamma' \vdash_\Sigma s : \sigma$ *and* $\Gamma' \vdash_\Sigma t : \sigma$*;*

(v) functional*:* $(\Gamma \vdash_\Sigma s : \sigma \longrightarrow_\beta \cup \longrightarrow_\eta t : \sigma)$ *implies* $(\Gamma \vdash_\Sigma s : \sigma) \succ (\Gamma \vdash_\Sigma t : \sigma)$.

Note that (iii) is indeed equivalent to $\Gamma\xi \vdash_\Sigma s\xi : \sigma\xi$ for all type substitutions ξ, therefore justifying separating (iii) from (ii).

We will often abuse notations by writing $(\Gamma \vdash_\Sigma s : \sigma \succ t : \tau)$ instead of $(\Gamma \vdash_\Sigma s : \sigma) \succ (\Gamma \vdash_\Sigma t : \tau)$, therefore sharing the environment Γ. This amounts to view the ordering as a relation on typed terms instead of a relation on judgements. We may even omit types and/or environments, considering the ordering as operating directly on terms, and write $\Gamma \vdash s \succ t$, $s : \sigma \succ t : \tau$, or $s \succ t$.

2.10 Termination of polymorphic higher-order rules

Theorem 2.18 *Let $\succeq$ be a polymorphic, higher-order reduction ordering and $R = \{\Gamma_i \vdash_\Sigma l_i \to r_i\}_{i\in I}$ be a higher-order rewrite system such that $\forall i \in I,\ (\Gamma_i \vdash_\Sigma l_i : \sigma_i) \succ (\Gamma_i \vdash_\Sigma r_i : \sigma_i)$, where σ_i is the principal type of l_i in Γ_i. Then the relation $\longrightarrow_R \cup \longrightarrow_\beta \cup \longrightarrow_\eta$ is strongly normalizing.*

Proof: Let $\Gamma \vdash_\Sigma s : \sigma$ and $\Gamma \vdash_\Sigma s \xrightarrow[\Gamma \vdash l\to r\in R]{p} t$. We assume without loss of generality that $\mathcal{V}ar(l) \cap \mathcal{D}om(\Gamma) = \emptyset$. By Lemma 2.6, $\Gamma_{s|_p} \vdash_\Sigma s|_p : \tau$, the actual type of $s|_p$ in the environment $\Gamma_{s|_p}$. By Definition 2.15(iii,iv) and Lemma 2.12, $s|_p = l\delta\gamma$ and $t|_p = r\delta\gamma$, where δ is a specialization and γ is type preserving. Therefore, $\Gamma_{s|_p} \vdash_\Sigma l\delta\gamma : \tau$. Since $\Gamma \cdot \mathcal{R}an(\delta\gamma) \subseteq \Gamma_{s|_p}$ by our assumption and Definition 2.15(ii) and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \subseteq \mathcal{V}ar(\Gamma)$ by Definition 2.14(i), then $\Gamma \cdot \mathcal{R}an(\delta\gamma) \vdash_\Sigma l\delta\gamma : \tau$ by applications of lemma 2.5. Since γ is type preserving, $\Gamma \cdot \mathcal{R}an(\delta) \vdash_\Sigma l\delta : \tau$. By Lemma 2.16, it follows that $\Gamma \cdot \mathcal{R}an(\delta) \vdash_\Sigma r\delta : \tau$, $\Gamma \cdot \mathcal{R}an(\delta\gamma) \vdash_\Sigma r\delta\gamma : \tau$, and $\Gamma \vdash_\Sigma t : \sigma$.

By polymorphism, $\Gamma \cdot \mathcal{R}an(\delta) \vdash_\Sigma l\delta : \tau \succ r\delta : \tau$. By stability, $\Gamma \cdot \mathcal{R}an(\delta\gamma) \vdash_\Sigma l\delta\gamma : \tau \succ r\delta\gamma : \tau$. By the previous remark that $\Gamma \cdot \mathcal{R}an(\delta\gamma) \subseteq \Gamma_{s|_p}$ and by compatibility, $\Gamma_{s|_p} \vdash_\Sigma l\delta\gamma : \tau \succ r\delta\gamma : \tau$. By monotonicity and Lemma 2.8, $\Gamma_{s|_p} \cdot \Gamma \vdash_\Sigma s[l\delta\gamma] : \sigma \succ s[r\delta\gamma] : \sigma$. By Definition 2.15(iii,iv), $\Gamma_{s|_p} \cdot \Gamma \vdash_\Sigma s : \sigma \succ t : \sigma$. By compatibility again, $\Gamma \vdash_\Sigma s : \sigma \succ t : \sigma$.

Finally, the case of a β- or an η-step follows from functionality. □

The polymorphic property of higher-order reduction orderings allows us to show termination in all monomorphic instances of the signature by means of a single comparison for each polymorphic rewrite rule. Polymorphic higher-order reduction orderings are therefore an appropriate tool in order to make termination proofs of polymorphic plain higher-order rewrite systems. In case of a monomorphic rewrite system, there is of course no need for a polymorphic ordering, even if the signature itself is polymorphic.

We are left with constructing polymorphic higher-order reduction orderings.

3 The Higher-Order Recursive Path Ordering

The higher-order recursive path ordering (HORPO) on higher-order terms is generated from three basic ingredients: a *type ordering*; a *precedence* on functions symbols; and a *status* for the function symbols. We describe these ingredients before defining the higher-order recursive path ordering, and study its properties, including strong normalization. We then consider a first set of examples which can all be run with our prototype implementation. Some proofs are omitted in this section, since they will be carried out in Section 4, in which a more advanced version of HORPO is studied.

3.1 The ingredients

- A quasi-ordering on types $\geq_{\mathcal{T}_\mathcal{S}}$ called *the type ordering* satisfying the following properties:
  1. *Well-foundedness*: $>_{\mathcal{T}_\mathcal{S}}$ is well-founded;
  2. *Arrow preservation*: $\tau \to \sigma =_{\mathcal{T}_\mathcal{S}} \alpha$ iff $\alpha = \tau' \to \sigma',\ \tau' =_{\mathcal{T}_\mathcal{S}} \tau$ and $\sigma =_{\mathcal{T}_\mathcal{S}} \sigma'$;
  3. *Arrow decreasingness*: $\tau \to \sigma >_{\mathcal{T}_\mathcal{S}} \alpha$ implies $\sigma \geq_{\mathcal{T}_\mathcal{S}} \alpha$ or $\alpha = \tau' \to \sigma', \tau' =_{\mathcal{T}_\mathcal{S}} \tau$ and $\sigma >_{\mathcal{T}_\mathcal{S}} \sigma'$;
  4. *Arrow monotonicity*: $\tau \geq_{\mathcal{T}_\mathcal{S}} \sigma$ implies $\alpha \to \tau \geq_{\mathcal{T}_\mathcal{S}} \alpha \to \sigma$ and $\tau \to \alpha \geq_{\mathcal{T}_\mathcal{S}} \sigma \to \alpha$;
  5. *Stability*: (i) $\sigma >_{\mathcal{T}_\mathcal{S}} \tau$ implies $\sigma\xi >_{\mathcal{T}_\mathcal{S}} \tau\xi$ for all type substitution ξ;
     (ii) $\sigma =_{\mathcal{T}_\mathcal{S}} \tau$ implies $\sigma\xi =_{\mathcal{T}_\mathcal{S}} \tau\xi$ for all type substitution ξ.

We show the existence of a type ordering generated by the above properties in Section 3.5.1, and give a particular type ordering in Section 5.1.

- A quasi-ordering $\geq_{\mathcal{F}}$ on $\mathcal{F}$, called the *precedence*, such that $>_{\mathcal{F}}$ is well-founded.
- A *status* $stat_f \in \{Mul, Lex\}$ for every symbol $f \in \mathcal{F}$, therefore partitioning $\mathcal{F}$ into $Mul \uplus Lex$. We say that f has a *multiset* or (left-to-right) *lexicographic* status if $f \in Mul$ and $f \in Lex$ respectively. We use a right-to-left status in one example.

3.2 The definition

The definition of the higher-order recursive path ordering builds upon Dershowitz's recursive path ordering (RPO) for first-order terms [16]. A major difference is that we do not compare terms, but terms together with their type in a given environment. A difficulty is that the environments in which two given terms are compared may change along recursive calls, and even change differently for the two recursively compared terms, which makes it impossible to factor the environment out. As a consequence, the ordering is defined on pairs of judgements $(\Gamma \vdash s : \sigma, \Sigma \vdash t : \tau)$ instead of on pairs of terms (s, t). Following Theorem 2.18, the starting comparison is of the form $\Gamma \vdash_\Sigma l : \sigma \succ \Gamma \vdash_\Sigma r : \sigma$ for some rule $\Gamma \vdash_\Sigma l \to r$, where σ is the principal type of l in Γ and a non-necessarily principal type of r in Γ. We are therefore using actual rather than principal types [1]. A second difficulty is that we need to compare subterms recursively, hence to have an appropriate typing judgement for a subterm in a term. Definition 2.7 provides the answer by giving us the actual type of a subterm in a term with respect to the typing judgement. In the sequel, all judgements $\Gamma \vdash_\Sigma s : \sigma$ we consider are therefore canonical, and σ is the actual type of s, allowing us to use the simpler notation $\vdash_\Sigma$ in place of the more precise $\vdash^c_\Sigma$.

Following the tradition, we consider equivalence classes of terms modulo α-conversion, using the syntactic equality symbol $=$ for the equivalence $=_\alpha$, and define our ordering $\succ_{horpo}$ by means of a set of rules, writing $\succeq_{horpo}$ for $\succ_{horpo} \cup =$. In contrast, the recursive path ordering contains a non-trivial equivalence allowing one to freely permute subterms below multiset function symbols [17]. Using here a richer equivalence relation would raise technical complications for no practical benefit.

The definition starts with 4 cases reproducing Dershowitz's recursive path ordering for first-order terms, with one main difference when higher-order terms are compared: the rules can also take care of higher-order terms in the arguments of the smaller side by having a corresponding bigger higher-order term in the arguments of the bigger side. This is redundant for first-order terms because of the subterm property. This idea is captured in the following *weak subterm property* used throughout the definition :

$$A = \forall v \in \overline{t}\ \ (\Gamma \vdash_\Sigma s : \sigma) \underset{horpo}{\succ} (\Sigma \vdash_\Sigma v : \rho) \text{ or } (\Gamma \vdash_\Sigma u : \theta) \underset{horpo}{\succ} (\Sigma \vdash_\Sigma v : \rho) \text{ for some } u \in \overline{s}$$

Cases 5 and 6 allow one to use the subterm case for applications and abstractions. In the abstraction case, we may need to rename the bound variable x of s to prevent confusing it with a free variable of t. Cases 7 and 8 are precedence cases for comparing a term headed by an algebraic symbol with a term headed by an application or an abstraction. Case 7 is crucial for the usefulness of the ordering. Cases 9 and 10 allow one to compare terms headed both by applications or both by lambdas. They are essential for monotonicity. Cases 11 and 12 include respectively β- and η-reductions in the ordering.

Definition 3.1 *Given two judgements* $\Gamma \vdash_\Sigma s : \sigma$ *and* $\Sigma \vdash_\Sigma t : \tau$*,*

$$(\Gamma \vdash_\Sigma s : \sigma) \underset{horpo}{\succ} (\Sigma \vdash_\Sigma t : \tau) \textit{ iff } \sigma \geq_{\mathcal{TS}} \tau \textit{ and}$$

[1] Using principal types would require that l and r have the same principal type in Γ to make sure that they are comparable. We have actually implemented both choices, and all our examples run on both implementations.

1. $s = f(\overline{s})$ *with* $f \in \mathcal{F}$, *and* $(\Gamma \vdash_\Sigma u : \theta) \underset{horpo}{\succeq} (\Sigma \vdash_\Sigma t : \tau)$ *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* A
3. $s = f(\overline{s})$ *and* $t = g(\overline{t})$ *with* $f =_\mathcal{F} g \in Mul$, *and* $(\Gamma \vdash_\Sigma \overline{s} : \overline{\sigma})\ (\underset{horpo}{\succ})_{mul}\ (\Sigma \vdash_\Sigma \overline{t} : \overline{\tau})$
4. $s = f(\overline{s})$ *and* $t = g(\overline{t})$ *with* $f =_\mathcal{F} g \in Lex$, *and* $(\Gamma \vdash_\Sigma \overline{s} : \overline{\sigma})\ (\underset{horpo}{\succ})_{lex}\ (\Sigma \vdash_\Sigma \overline{t} : \overline{\tau})$ *and* A
5. $s = @(s_1, s_2)$, *and* $(\Gamma \vdash_\Sigma s_1 : \rho \to \sigma) \underset{horpo}{\succeq} (\Sigma \vdash_\Sigma t : \tau)$ *or* $(\Gamma \vdash_\Sigma s_2 : \rho) \underset{horpo}{\succeq} (\Sigma \vdash_\Sigma t : \tau)$
6. $s = \lambda x : \alpha.u$ *with* $x \notin \mathcal{V}ar(t)$, *and* $(\Gamma \cdot \{x : \alpha\} \vdash_\Sigma u : \theta) \underset{horpo}{\succeq} (\Sigma \vdash_\Sigma t : \tau)$
7. $s = f(\overline{s})$ *with* $f \in \mathcal{F}$, $t = @(\overline{t})$ *is a partial left-flattening of* t, *and* A
8. $s = f(\overline{s})$ *with* $f \in \mathcal{F}$, $t = \lambda x : \alpha.v$ *with* $x \notin \mathcal{V}ar(v)$ *and* $(\Gamma \vdash_\Sigma s : \sigma) \underset{horpo}{\succ} (\Sigma \vdash_\Sigma v : \rho)$
9. $s = @(s_1, s_2)$, $t = @(\overline{t})$ *is a partial left-flattening of* t, *and* $\{(\Gamma \vdash_\Sigma s_1 : \theta \to \sigma), (\Gamma \vdash_\Sigma s_2 : \theta)\}(\underset{horpo}{\succ})_{mul}(\Sigma \vdash_\Sigma \overline{t} : \overline{\tau})$
10. $s = \lambda x : \alpha.u,\ t = \lambda x : \beta.v$, $\alpha =_{\mathcal{T}_\mathcal{S}} \beta$, *and* $(\Gamma \cdot \{x : \alpha\} \vdash_\Sigma u : \theta) \underset{horpo}{\succ} (\Sigma \cdot \{x : \beta\} \vdash_\Sigma v : \rho)$
11. $s = @(\lambda x : \alpha.u, v)$ *and* $(\Gamma \vdash_\Sigma u\{x \mapsto v\} : \sigma) \underset{horpo}{\succeq} (\Sigma \vdash_\Sigma t : \tau)$
12. $s = \lambda x : \alpha.@(u, x),\ x \notin \mathcal{V}ar(u)$ *and* $(\Gamma \vdash_\Sigma u : \sigma) \underset{horpo}{\succeq} (\Sigma \vdash_\Sigma t : \tau)$

The definition is recursive, and, apart from case 11, recursive calls operate on judgements whose terms are subterms of the term in the starting judgement. This ensures the well-foundedness of the definition, since the union of β-reduction with subterm is well-founded, by comparing pairs of argument terms in the well-founded compatible relation $(\longrightarrow_\beta \cup \rhd, \rhd)_{lex}$. This will actually be used as an inductive argument in many proofs to come. It must be stressed that multiset and lexicographic extensions are defined for arbitrary relations and preserve well-foundedness of arbitrary (well-founded) relations. As a consequence, our definition is inductive.

Carrying judgements systematically would often make it look awkward. Therefore, from now on, *we indulge forgetting judgements when comparing terms, leaving environments and types implicit when possible.*

Example 9 (Example 8 continued). We use here the existence of an ordering on types generated by the properties of the type ordering introduced in Section 3.1, and assume a multiset status for rec. The first rule succeeds immediately by case 1. For the second, we apply case 7, and need to show recursively that (i) $X \succeq_{horpo} X$, (ii) $s(x) \succ_{horpo} x$, and (iii) $rec(s(x), u, X) \succ_{horpo} rec(x, u, X)$.
(i) is clear. (ii) is by case 1. (iii) is by case 3, calling again recursively for $s(x) \succ_{horpo} x$.

Note that we have proved termination of Gödel's polymorphic recursor, for which the output type of rec is any given type. This is so because our ordering is polymorphic, as we will show. This example was already proved in [32], where polymorphism was claimed but neither formally defined nor proved.

We can of course now add some defining rules for sum and product for which we omit environments and types for bound variables which can be easily inferred by the reader:

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

The first two rules are easy work. For the third, we use the precedence $* >_{\mathcal{F}} rec$ to eliminate the rec operator. But the computation fails, since no subterm of $x*y$ can take care of the righthand side subterm $\lambda z_1 z_2.x+z_2$. We will come back to this example later, after having boosted the ordering. □

Example 10 We consider here lists of natural numbers. Let

$$\begin{array}{l} \mathcal{S} = \{List, \mathbb{N}\} \\ \mathcal{F} = \{nil : List,\ cons : \mathbb{N} \times List \Rightarrow List,\ map : List \times (\mathbb{N} \to \mathbb{N}) \Rightarrow List\} \end{array}$$

The rules for map are:

$$\begin{array}{rcrcl} \{X : \mathbb{N} \to \mathbb{N}\} & \vdash & map(nil, X) & \to & nil \\ \{x : \mathbb{N},\ l : List,\ X : \mathbb{N} \to \mathbb{N}\} & \vdash & map(cons(x,l), X) & \to & cons(@(X,x), map(l,X)) \end{array}$$

We use the ordering on types generated as previously by the properties of the type ordering and the additional equality $\mathbb{N} =_{\mathcal{T}_\mathcal{S}} List$. For the precedence, we take $map >_{\mathcal{F}} cons$, and $map \in Mul$ for the statuses.

The first rule is easily taken care of by case 1. For the second, since $map >_{\mathcal{F}} cons$, applying case 2 yields the subgoals $map(cons(x,l),X) \succ_{horpo} @(X,x)$ and $map(cons(x,l),X) \succ_{horpo} map(l,X)$. The latter is true by case 3, since $cons(x,l) \succ_{horpo} l$ by case 1. The first is by case 7, as X is an argument of the first term and $cons(x,l) \succ_{horpo} x$ by case 1. Note that we use the type comparison $List =_{\mathcal{T}_\mathcal{S}} \mathbb{N}$ in this computation.

Example 11 We now consider a specification of polymorphic lists. Let

$$\begin{array}{c} \mathcal{S}^{\forall} = \{\alpha\}; \mathcal{S} = \{List : * \Rightarrow *\}; \\ \mathcal{F} = \{\ nil : \alpha,\ cons : \alpha \times List(\alpha) \Rightarrow List(\alpha),\ map : List(\alpha) \times (\alpha \to \alpha) \Rightarrow List(\alpha)\ \} \end{array}$$

The rules for *map* become:

$$\begin{array}{rcrcl} \{X : \alpha \to \alpha\} & \vdash & map(nil, X) & \to & nil \\ \{x : \alpha,\ l : List(\alpha),\ X : \alpha \to \alpha\} & \vdash & map(cons(x,l), X) & \to & cons(@(X,x), map(l,X)) \end{array}$$

Letting again $map \in Mul$ and $map >_{\mathcal{F}} cons$, the first rule is taken care of by case 1 as previously. For the second, we need to set $List(\alpha) >_{\mathcal{T}_\mathcal{S}} \alpha$ for the subgoal $map(cons(x,l),X) \succ_{horpo} @(X,x)$. The computation goes then as previously. □

Important observations are the following:

- HORPO compares terms of comparable types, improving over [32] where terms of equivalent types only could be compared, as done in Example 10. Without type comparisons being based on an ordering instead of an equivalence, we could never allow a subterm case for terms headed by an application or by an abstraction. Subterm cases, however, must be controlled by the type comparison for the ordering to be well-founded.
- Another important improvement for practice is the inclusion of β- and η-reductions in the ordering via Cases 11 and 12. We will see several examples where it is used.

- Note that there is no precedence case for applications against abstractions and that partial-flattening is used in the righthand sides only, that is, in Cases 7 and 9, and indeed, our strong normalization proof does not go through if it is also used in the lefthand sides. left-flattening is essential for stability. The choice of a particular partial left-flattening in practice is considered in Section 5.
- HORPO is unfortunately not transitive, therefore explaining our statement in Theorem 3.2. Removing Cases 11 and 12, as well as left-flattening in Cases 7 and 9 yields a transitive ordering having the weak-subterm property. Some examples will indeed need using one step of transitivity, which raises some difficulties for the implementation addressed in Section 5.
- When the signature is first-order, Cases 1, 2, 3 and 4 of the definition of $\succ_{horpo}$ together reduce to the usual recursive path ordering for first-order terms, whose complexity is known to be in $O(n^2)$ for an input of size n. Inferring a precedence for comparing two given terms is also known to be NP-complete. This is not a problem for practice since signatures and terms are usually small. We have not investigated the complexity of HORPO, because an unbounded use of Case 11 may result in an arbitrary high complexity. We think however that both results still hold for HORPO when dropping Case 11 or restricting its use.

We now state the main result of this section, whose proof is the subject of the coming two subsections:

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

3.3 Candidate Terms

Because our strong normalization proof is based on Tait and Girard's reducibility technique, we need to associate to each type σ, actually to the equivalence class of σ modulo $=_{\mathcal{TS}}$, a set of terms $[\![\sigma]\!]$ closed under certain operations. In particular, if $s \in [\![\sigma \to \tau]\!]$ and $t \in [\![\sigma]\!]$, then the raw term $@(s,t)$ must belong to the set $[\![\tau]\!]$ even if it is not typable. In this section, we give a more liberal type system in which all raw terms needed in the strong normalization proof become typable *candidate terms*.

Variables:
$$\frac{x : \sigma \in \Gamma \quad \sigma' =_{\mathcal{TS}} \sigma}{\Gamma \vdash_\Sigma x :_C \sigma'}$$

Abstraction:
$$\frac{\Gamma \cdot \{x : \sigma\} \vdash_\Sigma t :_C \tau \quad \theta =_{\mathcal{TS}} \sigma}{\Gamma \vdash_\Sigma (\lambda x : \sigma.t) :_C \theta \to \tau}$$

Instantiation:
$$\frac{\Gamma \vdash_\Sigma s :_C \sigma \quad \xi \text{ some type substitution of domain } \mathcal{V}ar(\sigma) \quad \sigma\xi =_{\mathcal{TS}} \tau}{\Gamma \vdash_\Sigma s :_C \tau}$$

Application:
$$\frac{\Gamma \vdash_\Sigma s :_C \sigma \quad \Gamma \vdash_\Sigma t :_C \tau \quad \xi \text{ most general unifier of } \alpha \to \beta = \sigma \wedge \alpha = \tau \quad \beta\xi =_{\mathcal{TS}} \tau}{\Gamma \vdash_\Sigma @(s,t) :_C \tau}$$

Functions:
$$\frac{f : \sigma_1 \times \ldots \times \sigma_n \to \sigma \in \mathcal{F} \quad \Gamma \vdash_\Sigma t_1 :_C \tau_1 \ \ldots \ \Gamma \vdash_\Sigma t_n :_C \tau_n \quad \xi \text{ most general unifier of } \sigma_1 = \tau_1 \wedge \ldots \wedge \sigma_n = \tau_n \quad \tau =_{\mathcal{TS}} \sigma\xi}{\Gamma \vdash_\Sigma f(t_1, \ldots, t_n) :_C \tau}$$

Figure 2. Typing judgements for candidate terms

Definition 3.3 *A raw term is a* candidate terms *if it is typable in the type system of Figure 2.*

The set of types of a typable candidate term of type σ is a union of type equivalence classes modulo $=_{\mathcal{TS}}$:

Lemma 3.4 *$\forall \sigma\tau$ such that $\sigma =_{\mathcal{T}_S} \tau$, $\Gamma \vdash_\Sigma s :_C \sigma$ iff $\Gamma \vdash_\Sigma s : \tau$.*

This allows us to talk about *the types* of a candidate term up to type equivalence. Note finally five easy, important properties of candidate terms:

1. Every term is a candidate term;
2. Typing Lemmas 2.4, 2.5, 2.6, 2.8 and 2.13.1 hold for candidate terms;
3. HORPO applies to candidate terms as well as to terms, by keeping the same definition;
4. Because β-reduction is type preserving, the set of candidate terms is closed under β-reductions;
5. Because β-reduction is strongly normalizing for a typed λ-calculus with arbitrary constants, the set of candidate terms is well-founded with respect to β-reductions (for the argument, consider a signature $\mathcal{F}'$ in which $f : \sigma'_1 \times \ldots \times \sigma'_n \Rightarrow \sigma' \in \mathcal{F}'$ provided $f : \sigma_1 \times \ldots \times \sigma_n \Rightarrow \sigma \in \mathcal{F}$ with $\sigma'_i =_{\mathcal{T}_S} \sigma_i$ for every $i \in [1..n]$ and $\sigma' =_{\mathcal{T}_S} \sigma$).

To make sense of the second observation, we need to show that $\succ_{horpo}$, when comparing candidate terms, is compatible with type equivalence.

Lemma 3.5 *Assume given candidate terms s, t and types σ, σ', τ, τ' such that $s :_C \sigma \succ_{horpo} t :_C \tau$, $\sigma =_{\mathcal{T}_S} \sigma'$ and $\tau =_{\mathcal{T}_S} \tau'$. Then $s :_C \sigma' \succ_{horpo} t :_C \tau'$.*

Proof: By Lemma 3.4, $s :_C \sigma'$ and $t :_C \tau'$, hence the statement makes sense. Now, since $=_{\mathcal{T}_S} \subseteq \geq_{\mathcal{T}_S}$, then $\sigma \geq_{\mathcal{T}_S} \tau$ implies $\sigma' \geq_{\mathcal{T}_S} \tau'$ by transitivity.

The proof of the ordering statement proceeds by induction on $(\longrightarrow_\beta \cup \rhd, \rhd)_{lex}$, and by case on the definition of the ordering, using for each case the induction hypothesis together with Lemma 3.4. □

3.4 Ordering properties of HORPO

Because the strong normalization proof requires using candidate terms, two results of this section, monotonicity and stability, which are used in the strong normalization proof, must be stated and proved for both terms and candidate terms. Since terms are closed under taking subterms, proofs are indeed essentially the same in both cases. We do them for candidate terms.

Lemma 3.6 (Stability) *$\succ_{horpo}$ is stable for candidate terms and for terms.*

Proof: Let $\Gamma \vdash_\Sigma s :_C \sigma \succ_{horpo} t :_C \tau$ and γ be a type preserving substitution compatible with Γ. By Lemma 2.13.1, $\Gamma \cdot \mathcal{R}an(\gamma) \vdash_\Sigma s\gamma :_C \sigma$ and $\Gamma \cdot \mathcal{R}an(\gamma) \vdash_\Sigma t\gamma :_C \tau$. We show that $\Gamma \cdot \mathcal{R}an(\gamma) \vdash_\Sigma s\gamma :_C \sigma \succ_{horpo} t\gamma :_C \tau$ by induction on $(\longrightarrow_\beta \cup \rhd, \rhd)_{lex}$. Since type preserving substitutions preserve types, we will from now on only consider the term comparisons of the ordering and omit all references to types and judgements. There are 12 cases according to the definition:

1. If $s \succ_{horpo} t$ by case 1, then $s_i \succeq_{horpo} t$, and by induction hypothesis $s_i\gamma \succeq_{horpo} t\gamma$, and therefore, $s\gamma \succ_{horpo} t\gamma$ by case 1.
2. If $s \succ_{horpo} t$ by case 2, then $s = f(\overline{s})$, $t = g(\overline{t})$, $f >_{\mathcal{F}} g$ and for all $t_i \in \overline{t}$ either $s \succ_{horpo} t_i$ or $s_j \succeq_{horpo} t_i$ for some $s_j \in \overline{s}$. By induction hypothesis, for all $t_i\gamma \in \overline{t\gamma}$ either $s\gamma \succ_{horpo} t_i\gamma$ or $s_j\gamma \succeq_{horpo} t_i\gamma$ for some $s_j\gamma \in \overline{s\gamma}$. Therefore, $s\gamma = f(\overline{s\gamma}) \succ_{horpo} g(\overline{t\gamma}) = t\gamma$ by case 2.

3. If $s \succ_{horpo} t$ by case 3, then $s = f(\overline{s})$, $t = g(\overline{t})$, $f =_{\mathcal{F}} g$, $f, g \in Mul$ and $\overline{s}(\succ_{horpo})_{mul}\overline{t}$. By induction hypothesis $\overline{s\gamma}(\succ_{horpo})_{mul}\overline{t\gamma}$, and hence $s\gamma \succ_{horpo} t\gamma$ by case 3.

4. If $s \succ_{horpo} t$ by case 4, then $s = f(\overline{s})$, $t = g(\overline{t})$, $f =_{\mathcal{F}} g$, $f, g \in Lex$, $\overline{s}(\succ_{horpo})_{lex}\overline{t}$, and for all $t_i \in \overline{t}$ either $s \succ_{horpo} t_i$ or $s_j \succeq_{horpo} t_i$ for some $s_j \in \overline{s}$. By induction hypothesis $\overline{s\gamma}(\succ_{horpo})_{lex}\overline{t\gamma}$, and as in the precedence case, by induction hypothesis, for all $t_i\gamma \in \overline{t\gamma}$ either $s\gamma \succ_{horpo} t_i\gamma$ or $s_j\gamma \succeq_{horpo} t_i\gamma$ for some $s_j\gamma \in \overline{s\gamma}$. Therefore $s\gamma \succ_{horpo} t\gamma$ by case 4.

5. If $s \succ_{horpo} t$ by case 5, the reasoning is similar to Case 1.

6. If $s \succ_{horpo} t$ by case 6, then $s = \lambda x.u$ and $u \succeq_{horpo} t$. Let γ a substitution of domain $\mathcal{V}ar(s) \cup \mathcal{V}ar(t)$, hence $x \notin \mathcal{D}om(\gamma)$ by assumption on x. By induction hypothesis, $u\gamma \succeq_{horpo} t\gamma$, hence $s\gamma = \lambda x.u\gamma \succ_{horpo} t\gamma$ by case 6.

7. If $s \succ_{horpo} t = @(\overline{t})$ by case 7, then for every $t_i \in \overline{t}$, either $s \succ_{horpo} t_i$, and $s\gamma \succ_{horpo} t_i\gamma$ by induction hypothesis, or $s_j \succeq_{horpo} t_i$ for some $s_j \in \overline{s}$, and $s_j\gamma \succ_{horpo} t_i\gamma$ by induction hypothesis. Therefore, since $@(\overline{t\gamma})$ is a partial left-flattening of $t\gamma$, $s\gamma \succ_{horpo} t\gamma$ by case 7.

8. If $s \succ_{horpo} t$ by Case 8, then $t = \lambda x.v$ with $x \notin \mathcal{V}ar(v)$ and $s \succ_{horpo} v$. By induction hypothesis, $s\gamma \succ_{horpo} v\gamma$ for every substitution γ of domain $\mathcal{V}ar(v) \setminus \{x\}$ such that $x \notin \mathcal{V}ar(v\gamma)$, hence $s\gamma \succ_{horpo} t\gamma = \lambda x.v\gamma$ by Case 8.

9. If $s = @(s_1, s_2) \succ_{horpo} t = @(\overline{t})$ by case 9, then the proof goes as in case 2, concluding by case 9.

10. If $s \succ_{horpo} t$ by case 10, then $s = \lambda x.u$, $t = \lambda x.v$ and $u \succ_{horpo} v$. By induction hypothesis $u\gamma \succ_{horpo} v\gamma$. Assuming that $x \notin \mathcal{D}om(\gamma)$, then $s\gamma = \lambda x.u\gamma \succ_{horpo} \lambda x.v\gamma = t\gamma$ by case 10.

11. If $s = \succ_{horpo} t$ by case 11, then the property holds by stability of β-reduction.

12. If $s = \succ_{horpo} t$ by case 12, then the property holds by stability of η-reduction. □

Lemma 3.7 (Polymorphism) *$\succ_{horpo}$ is polymorphic for candidate terms and for terms.*

Proof: Let $\Gamma \vdash_\Sigma s :_C \sigma \succ_{horpo} t :_C \tau$ and δ be a specialization compatible with Γ. By Lemma 2.13.2, $\Gamma \cdot \mathcal{R}an(\delta) \vdash_\Sigma s\delta :_C \sigma\xi_\delta$ and $\Gamma \cdot \mathcal{R}an(\delta) \vdash_\Sigma t\delta :_C \tau\xi_\delta$. We show that $\Gamma \cdot \mathcal{R}an(\delta) \vdash_\Sigma s\delta :_C \sigma\xi_\delta \succ_{horpo} t\delta :_C \tau\xi_\delta$ as before, by induction on $(\longrightarrow_\beta \cup \rhd, \rhd)_{lex}$. Since the type ordering is stable, $\sigma \geq_{\mathcal{TS}} \tau$ implies $\sigma\xi_\delta \geq_{\mathcal{TS}} \tau\xi_\delta$ and the proof continues as previously, using the stability of the type ordering for each recursive comparison. □

Lemma 3.8 (Monotonicity) *$\succ_{horpo}$ is monotonic for candidate terms and for terms.*

Proof: Omitting the environment, we prove that $s :_C \sigma \succ_{horpo} t :_C \sigma$ implies $u[s] :_C \tau \succ_{horpo} u[t] :_C \tau$ for all $u[x : \sigma] :_C \tau$.

We proceed by induction on the size of u. If u is empty it holds. Otherwise there are three cases:

- $u[x : \sigma]$ is of the form $u'[f(\ldots, x : \sigma, \ldots)]$ for some function symbol f. If $f \in Mul$, then $f(\ldots s \ldots) :_C \theta \succ_{horpo} f(\ldots t \ldots) :_C \theta$ follows by Case 3, and we conclude by induction since u' is smaller than u. Otherwise, $f \in Lex$, hence $\{\ldots s \ldots\}(\succ_{horpo})_{lex}\{\ldots t \ldots\}$ and, for every $v \in \{\ldots t \ldots\}$, there exists $u \in \{\ldots s \ldots\}$ such that $u \succeq_{horpo} v$. Therefore, $f(\ldots s \ldots) \succ_{horpo} f(\ldots t \ldots)$ by Case 4 and we conclude by induction hypothesis as before.

- $u[x:\sigma]$ is of the form $u'[@(u, x:\sigma)]$ (resp. $u'[@(x:\sigma, u)]$). Then, $@(s,u):\theta \succ_{horpo} @(t,u):\theta$ (resp. $@(u,s):\theta \succ_{horpo} @(u,t):\theta$) follows by Case 9. We conclude by induction hypothesis.
- $u[x:\sigma]$ is of the form $u'[\lambda y.x]$. The result follows similarly by Case 10. □

From Lemmas 3.8 and 3.5, we easily get a generalized monotonicity property of $\succ_{horpo}$ for candidate terms of equivalent type. In the rest of the paper we will make use of this generalized monotonicity property by referring to Lemma 3.8. Note also that the monotonicity property of $\succ_{horpo}$ implies the monotonicity of the ordering $\succeq_{horpo}$ on terms of equivalent type.

A key usual consequence of monotonicity is that every subterm of a strongly normalizable term is itself strongly normalizable. This is true when decreasing sequences of terms are type preserving, since an infinite decreasing sequence originating in a subterm can be lifted in an infinite decreasing sequence originating in the superterm. This becomes false otherwise, since typing prevents lifting a decreasing sequence from a subterm to its superterm.

Lemma 3.9 *$\succ_{horpo}$ is compatible.*

This is so because types, hence comparisons, are preserved by the use of a compatible environment.

Lemma 3.10 *$\succ_{horpo}$ is functional.*

Proof: By definition and monotonicity. □

Although the above proofs are slightly more difficult technically than the usual proofs for the recursive path ordering, they follow the same kind of pattern (polymorphism was actually never considered before). This contrasts with the proof of strong normalization to come.

3.5 Strong Normalization

In this section, we consider the well-foundedness of the strict ordering $(\succ_{horpo})^+$, that is, equivalently, the strong normalization of the rewrite relation defined by the rules $s \longrightarrow t$ such that $s \succ_{horpo} t$. For the recursive path ordering, well-foundedness follows from Kruskal's tree theorem. Since we do not know of any non-trivial extension of Kruskal's tree theorem for higher-order terms that includes β-reductions (and wasted much of our time in looking for an appropriate one), we adopt a completely different method, the computability predicate proof method of Tait and Girard. To our knowledge, the use of this method for proving well-foundedness of a recursively defined relation is original. This proof method will suggest an important improvement of our ordering discussed in Section 4.

A proof based on that method starts by defining for each type σ, a set of terms called the computability predicate $[\![\sigma]\!]$. Terms in $[\![\sigma]\!]$ are said to be computable. So are substitutions made of computable terms. In practice, $[\![\sigma]\!]$ can be defined by the properties it should satisfy. The most important one is that computable terms must be strongly normalizable. Given an arbitrary computable, hence strongly normalizable substitution γ, one can then build an induction argument to prove that, for an arbitrary term t, the term $t\gamma$ is computable. The identity substitution being computable, we then conclude that t is computable, hence strongly normalizable. See [22] for a detailed exposition of the method in case of system F, and [20] for a discussion about the different possibilities for defining computability predicates in practice.

3.5.1 Type orderings

Two lemmas needed to justify our coming construction of the candidate interpretations: preservation of groundness first; second, any type ordering is included into an ordering enjoying the subterm property.

Lemma 3.11 *Assume σ is ground and $\sigma \geq_{\mathcal{TS}} \tau$. Then τ is ground as well.*

Proof: Because the type ordering is both stable and well-founded. □

Lemma 3.12 *Let $\geq_{\mathcal{TS}}$ be a quasi-ordering on types such that $>_{\mathcal{TS}}$ is well-founded, arrow monotonic and arrow preserving. Then, the relation $\geq_{\mathcal{TS}}^{\rightarrow} = (\geq_{\mathcal{TS}} \cup \rhd_{\rightarrow})^*$ is a well-founded quasi-ordering on types extending $\geq_{\mathcal{TS}}$ and $\rhd_{\rightarrow}$, whose equivalence coincides with $=_{\mathcal{TS}}$.*

Proof: First, we show that $\rhd_{\rightarrow}$ and $\geq_{\mathcal{TS}}$ commute, i.e that $\geq_{\mathcal{TS}} \cdot \rhd_{\rightarrow} \subseteq \rhd_{\rightarrow} \cdot \geq_{\mathcal{TS}}$. Assuming first that $\tau \rightarrow \sigma \rhd_{\rightarrow} \sigma \geq_{\mathcal{TS}} \rho$, then, by arrow monotonicity, $\tau \rightarrow \sigma \geq_{\mathcal{TS}} \tau \rightarrow \rho \rhd_{\rightarrow} \rho$. Assuming now that $\tau \rightarrow \sigma \rhd_{\rightarrow} \tau$ yields a similar computation. By transitivity of $\geq_{\mathcal{TS}}$, it follows that $\geq_{\mathcal{TS}}^{\rightarrow} = \geq_{\mathcal{TS}} \cdot \rhd_{\rightarrow}^*$.

We show now that $=_{\mathcal{TS}} = =_{\mathcal{TS}}^{\rightarrow}$. Assume that $\sigma \geq_{\mathcal{TS}} \rho_1 \rhd_{\rightarrow}^n \tau \geq_{\mathcal{TS}} \rho_2 \rhd_{\rightarrow}^m \sigma$ for types ρ_1 and ρ_2, and natural numbers n, m. By applying commutation twice followed by transitivity of $\geq_{\mathcal{TS}}$, we get $\sigma \rhd_{\rightarrow}^{m+n} \theta \geq_{\mathcal{TS}} \sigma$. Therefore, $\theta = u[\sigma]_p$ for some context u and position p with $|p| = m + n$, such that there is an arrow in u at each position $q < p$. Assume that $m + n > 0$. By arrow preservation $\sigma \neq_{\mathcal{TS}} u[\sigma]_p$, hence $\sigma >_{\mathcal{TS}} u[\sigma]_p$. By arrow monotonicity, $u[\sigma] \geq_{\mathcal{TS}} u[u[\sigma]]$ and by arrow preservation again, $u[\sigma] >_{\mathcal{TS}} u[u[\sigma]]$, and so on, resulting in an infinite sequence $\sigma >_{\mathcal{TS}} u[\sigma] >_{\mathcal{TS}} u[u[\sigma]] >_{\mathcal{TS}} \ldots$ which contradicts the well-foundedness of $>_{\mathcal{TS}}$. The property follows.

Since $\tau \rightarrow \sigma \geq_{\mathcal{TS}}^{\rightarrow} \tau$ and $\tau \rightarrow \sigma \geq_{\mathcal{TS}}^{\rightarrow} \sigma$ by definition of $\geq_{\mathcal{TS}}^{\rightarrow}$, and $=_{\mathcal{TS}} = =_{\mathcal{TS}}^{\rightarrow}$, arrow preservation implies that $\tau \rightarrow \sigma >_{\mathcal{TS}}^{\rightarrow} \tau$ and $\tau \rightarrow \sigma >_{\mathcal{TS}}^{\rightarrow} \sigma$.

We are left with well-foundedness. From $=_{\mathcal{TS}} = =_{\mathcal{TS}}^{\rightarrow}$ and $\geq_{\mathcal{TS}}^{\rightarrow} = \geq_{\mathcal{TS}} \cdot \rhd_{\rightarrow}^*$, it follows that $>_{\mathcal{TS}}^{\rightarrow} \subseteq >_{\mathcal{TS}} \cup \geq_{\mathcal{TS}} \cdot \rhd_{\rightarrow}^+$. We show that the latter is well-founded. Since $>_{\mathcal{TS}}$ is well-founded and $>_{\mathcal{TS}} \subseteq \geq_{\mathcal{TS}}$, any infinite sequence with $(>_{\mathcal{TS}} \cup \geq_{\mathcal{TS}} \cdot \rhd_{\rightarrow}^+)$ is an infinite sequence with $\geq_{\mathcal{TS}} \cdot \rhd_{\rightarrow}^+$. Commutation and well-foundeness of $\rhd_{\rightarrow}$ yield the contradiction. □

3.5.2 Candidate interpretations

Again, this section refers to candidate terms, rather than to candidate judgements. Computability of candidate terms of variable type is reduced to the computability of candidate terms of ground type by type instantiation. Our definition of computability for candidate terms of a ground type is standard, but we must make sure that it is compatible with the equivalence $=_{\mathcal{TS}}$ on types. Technically, we denote by $[\![\sigma]\!]$ the computability predicate of the type σ, which, by construction, will be equal to the predicate $[\![\tau]\!]$ for any type $\tau =_{\mathcal{TS}} \sigma$. Without loss of generality, we assume a global environment for variables.

Definition 3.13 *The family of* candidate interpretations $\{[\![\sigma]\!]\}_{\sigma \in \mathcal{TS}}$ *is a family of subsets of the set of candidates whose elements are the least sets satisfying the following properties:*

(i) If σ is a ground data type, then $s :_C \sigma \in [\![\sigma]\!]$ iff $t \in [\![\tau]\!]$ for all $t :_C \tau$ such that $s \succ_{horpo} t$

(ii) If σ is a ground functional type $\rho \rightarrow \tau$ then $s \in [\![\sigma]\!]$ iff $@(s,t) \in [\![\tau]\!]$ for all $t :_C \in [\![\rho]\!]$;

(iii) If σ is not ground, then $s \in [\![\sigma]\!]$ iff $s \in [\![\sigma\xi]\!]$ for all ground type substitutions ξ.

A candidate term s of type σ is said to be computable *if $s \in [\![\sigma]\!]$. A vector $\overline{s}$ of terms of type $\overline{\sigma}$ is computable iff so are all its components. A (candidate) term substitution γ is computable if all candidate terms in $\{x\gamma \mid x \in \mathcal{D}om(\gamma)\}$ are computable.*

Case (iii) in the definition allows us to eliminate non-ground types by appropriate type instantiations: all properties of candidate interpretations proved for ground types can then be easily lifted to non-ground ones. This is so because Cases (i) and (ii) of the definition refer to ground types only: in Case (ii), ρ and τ are ground since $\rho \to \tau$ is ground; in Case (i), τ is ground by Lemma 3.11. Therefore, $[\![\sigma]\!]$ is a subset of the set of candidate terms of type σ recursively defined in terms of itself in Case (i) when $\tau =_{\mathcal{T}_\mathcal{S}} \sigma$, and of other sets $[\![\tau]\!]$, in Case (i) and (ii) with $\sigma >_{\mathcal{T}_\mathcal{S}} \tau$, and $[\![\rho]\!]$ with $\sigma >^{\rightarrow}_{\mathcal{T}_\mathcal{S}} \rho$. Therefore, our definition splits into two: a definition of candidate interpretations on ground types, based on a lexicographic combination of an induction on the well-founded type ordering $>^{\rightarrow}_{\mathcal{T}_\mathcal{S}}$ (which includes $>_{\mathcal{T}_\mathcal{S}}$), and a fixpoint computation for data types; a definition of candidate interpretations for non-ground types which reduces to the definition on ground types.

Instead of our indexed family of sets, we could consider the function $F : \mathcal{T}_\mathcal{S} \longrightarrow 2^{\mathcal{T}}$ such that $F(\sigma) = [\![\sigma]\!]$. This function F is defined by induction on types (using $>^{\rightarrow}_{\mathcal{T}_\mathcal{S}}$). Besides, for each date type σ, $F(\sigma)$ is defined by a fixpoint computation (Case (i) with $\tau =_{\mathcal{T}_\mathcal{S}} \sigma$). Since Case (i) does not involve any negation, it is monotonic with respect to set inclusion, therefore ensuring the existence of a least fixpoint.

The choice of a particular formulation for the computability predicate is entirely driven by the *computability properties* one needs. Most of these are proved for ground types by an "induction on the definition of the candidate interpretations", by which we mean an outer induction on the type ordering $>^{\rightarrow}_{\mathcal{T}_\mathcal{S}}$ followed by an inner induction on the fixpoint computation. Since the ordering computations involve subterms, a potential difficulty is that a term of a ground type may have subterms of a non-ground type. Fortunately, type comparisons will imply the type-groundness property when needed.

We denote by $\mathcal{T}^{min}_\mathcal{S}$ the set of ground types which are minimal with respect to $>^{\rightarrow}_{\mathcal{T}_\mathcal{S}}$.

Lemma 3.14 *Assuming that $\mathcal{S}$ contains a constant, then $\mathcal{T}^{min}_\mathcal{S}$ is a non-empty set of data types.*

Proof: Because $>^{\rightarrow}_{\mathcal{T}_\mathcal{S}}$ is well-founded and arrow types cannot be minimal in $>^{\rightarrow}_{\mathcal{T}_\mathcal{S}}$. □

Preservation of data types follows easily from arrow preservation and stability:

Lemma 3.15 *Assume that $\sigma =_{\mathcal{T}_\mathcal{S}} \tau$ and σ is a data type, then τ is a data type as well.*

The following property is a clear consequence of the definition, Lemma 3.4 and arrow preservation:

Lemma 3.16 *Assume $\sigma =_{\mathcal{T}_\mathcal{S}} \tau$. Then $[\![\sigma]\!] = [\![\tau]\!]$.*

In the sequel, we assume that functional types are in canonical form and that $n > 0$ in $\sigma = \sigma_1 \to \ldots \to \sigma_n \to \tau$.

We first give the properties of the interpretations. Properties (i) to (v) are standard. Recall that a term is neutral if it is not an abstraction. Property (vi) is adapted from the simple type discipline, in which case the property is true of all basic types. Property (vii) is void when the algebraic signature is empty. It appeared first in [28].

Property 3.17 (Computability Properties)

(i) Every computable term is strongly normalizable;
(ii) Assuming that s is computable and $s \succeq_{horpo} t$, then t is computable;
(iii) A neutral term s is computable iff t is computable for every t such that $s \succ_{horpo} t$;
(iv) If $\bar{t}$ be a vector of computable terms such that $@(\bar{t})$ is a candidate term, then $@(\bar{t})$ is computable;
(v) $\lambda x : \sigma.u$ is computable iff $u\{x \mapsto w\}$ is computable for every computable term $w :_C \sigma$;
(vi) Let $s :_C \sigma \in \mathcal{T}^{min}_\mathcal{S}$. Then s is computable iff it is strongly normalizable.
(vii) Let $f : \overline{\sigma} \to \tau \in \mathcal{F}$ and $s = f(\overline{s}) :_C \sigma$. Then s is computable if $\overline{s} :_C \overline{\sigma}$ is computable.

Our definition does not explicitly state that variables are computable. Variables of a data type are of course computable by definition (item (i)) since they have no reduct. But variables of a functional type will be computable by computability property (iii). This will actually forbid us to prove the computability properties (i), (ii) and (iii) separately. A possible alternative would be to modify the definition of the computability predicates by adding the property that variables of a functional type are computable. This would make the properties (i), (ii) and (iii) independent to the price of other complications.

Proof:

- Property (iv).

  Straightforward induction on the length of $\bar{t}$ and use of Case (ii) of the definition.

- Properties (i), (ii), (iii).

  Note first that the only if part of property (iii) is property (ii). We are left with (i), (ii) and the if part of (iii) which we now spell out as follows:

  Given a ground type σ and a candidate term s such that $s :_C \sigma \in [\![\sigma]\!]$, we prove by induction on the definition of $[\![\sigma]\!]$ that

  (i) s is strongly normalizable;

  (ii) $\forall t :_C \tau$ such that $s \succ_{horpo} t$, t is computable;

  (iii) $\forall u :_C \sigma$ neutral, u is computable if $w :_C \theta \in [\![\theta]\!]$ for every w such that $u \succ_{horpo} w$.

  Since $s \succ_{horpo} t$ and $u \succ_{horpo} w$, it follows from the definition of the ordering that $\sigma \geq_{\mathcal{T}_S} \tau$ and $\sigma \geq_{\mathcal{T}_S} \theta$, implying that τ and θ are ground by Lemma 3.11. We prove each property in turn, distinguishing in each case whether σ is a data type or functional.

  (i) (a) Assume first that σ is a data type. All reducts of s are computable by definition of the interpretations, hence strongly normalizable by induction hypothesis, implying that s is strongly normalizable.

  (b) Assume now that $\sigma = \theta \to \tau$, and let $s_0 = s :_C \sigma = \theta_0 \succ_{horpo} s_1 :_C \theta_1 \ldots \succ_{horpo} s_n :_C \theta_n \succ_{horpo} \ldots$ be a derivation issuing from s. Note that θ_i must be ground by Lemma 3.11. Hence $s_n \in [\![\theta_n]\!]$ by assumption for $n = 0$ and repeated applications of induction property (ii) for $n > 0$. Such derivations are of the following two kinds:

  i. $\sigma >_{\mathcal{T}_S} \theta_i$ for some i, in which case s_i is strongly normalizable by induction hypothesis, hence the derivation issuing from s is finite;

  ii. $\theta_n =_{\mathcal{T}_S} \sigma$ for all n, in which case $\{@(s_n, y :_C \sigma_1) :_C \sigma_2 \to \ldots \sigma_n \to \tau\}_n$ is a sequence of candidate terms of ground types which is strictly decreasing with respect to $\succ_{horpo}$ by monotonicity. Since $\sigma >^{\rightarrow}_{\mathcal{T}_S} \sigma_1$, $y :_C \sigma_1$ is computable by induction hypothesis (iii), hence, by definition, $@(s_n, y)$ is computable. By induction hypothesis (i), the above sequence is finite, implying that the starting sequence itself is finite.

  (ii) (a) Assume that σ is a data type. The result holds by definition of the candidate interpretations.

  (b) Let $\sigma = \theta \to \rho$, hence θ and ρ are ground. By arrow preservation and decreasingness properties, there are two cases:

  i. $\rho \geq_{\mathcal{T}_S} \tau$. Since s is computable, $@(s, u)$ is computable for every $u \in [\![\theta]\!]$. Let $y :_C \theta$. By induction hypothesis (iii), $y \in [\![\theta]\!]$, hence $@(s, y)$ is computable. Since $@(s, y) :_C \rho \succ_{horpo} t :_C \tau$ by case 5 of the definition, t is computable by induction hypothesis (ii).

ii. $\tau = \theta' \to \rho'$, hence θ' and ρ' are ground, with $\theta =_{\mathcal{I}_S} \theta'$ and $\rho \geq_{\mathcal{I}_S} \rho'$. Since s is computable, given $u \in [\![\theta]\!]$, then $@(s,u) \in [\![\rho]\!]$, hence, by induction hypothesis (ii) $@(t,u) \in [\![\rho']\!]$. Since $[\![\theta]\!] = [\![\theta']\!]$ by Lemma 3.16, $t \in [\![\tau]\!]$ by definition of the interpretations.

(iii) (a) Assume that σ is a data type. The result holds by definition of the candidate interpretations.

(b) Assume now that $\sigma = \sigma_1 \to \ldots \to \sigma_n \to \tau$, where $n > 0$ and τ is a data type. Then, $\sigma_1, \ldots, \sigma_n, \tau$ are ground types. By property (iv) u is computable if $@(u, u_1, \ldots, u_n)$ is computable for arbitrary terms $u_1 \in [\![\sigma_1]\!], \ldots, u_n \in [\![\sigma_n]\!]$ which are strongly normalizable by induction property (i). By definition, as τ is a data type, $@(u, u_1, \ldots, u_n)$ is computable iff so are all its reducts.

We prove by induction on the multiset of computable terms $\{u_1, \ldots, u_n\}$ ordered by $(\succeq_{horpo})_{mul}$ the property (H) stating that all terms w strictly smaller than $@(u, u_1, \ldots, u_i)$ in $\succ_{horpo}$ are computable. Remark that w has a ground type by definition of the ordering and Lemma 3.11. Taking $i = n$ yields the desired property, implying that u is computable. If $i = 0$, terms strictly smaller than u are computable by assumption. For the general case, let $i = (j+1) \leq n$. We need to consider all terms w strictly smaller than $@(@(u, u_1, \ldots, u_j), u_{j+1})$. Since u is neutral, hence is not an abstraction, there are two possible cases:

i. $@(@(u, u_1, \ldots, u_j), u_{j+1}) \succ_{horpo} w$ by Case 5. There are again two possibilities:
   - $@(u, u_1, \ldots, u_j) \succeq_{horpo} w$, and therefore $@(u, u_1, \ldots, u_j) \succ_{horpo} w$ for type reason since w is also a reduct of $@(u, u_1, \ldots, u_{j+1})$. We then conclude by induction hypothesis (H).
   - $u_{j+1} \succeq_{horpo} w$. We conclude by assumption and induction property (ii).

ii. $@(@(u, u_1, \ldots, u_j), u_{j+1}) \succ_{horpo} w$ by Case 9, hence $w = @(\overline{w})$. By definition of the multiset extension and for type reasons, there are the following two possibilities:
   - for all $v \in \overline{w}$, either $@(u, u_1, \ldots, u_j) \succ_{horpo} v$, and v is computable by induction hypothesis (H), or $u_{j+1} \succeq_{horpo} v$, in which case v is computable by assumption and induction property (ii). It follows that w is computable by Property 3.17 (iv).
   - $w_1 = @(u, u_1, \ldots, u_j)$ and $u_{j+1} \succ_{horpo} w_2$, implying that w_2 is computable by assumption and induction property (ii). By induction property (H), all reducts of w are computable. Since w is an application hence is neutral, it is computable by induction property (iii).

As a consequence, all reducts of $@(u, u_1, \ldots, u_n)$ are computable and we are done. □

- Property (v), assuming that the type of $\lambda x : \sigma.u$ is ground. In particular, σ is a ground type, as well as the type of u.

  The only if part is property (ii) together with the definition of computability. For the if part, we prove that $@(\lambda x.u, w)$ is computable for an arbitrary computable w of type σ such that $u\{x \mapsto w\}$ is computable, implying that $\lambda x.u$ is computable by Definition 3.13 (ii).

  Since variables are computable by property (iii), $u = u\{x \mapsto x\}$ is computable by assumption. By property (i), u and w are strongly normalizable, hence they are strongly normalizable by property (i). We can now prove that $@(\lambda x.u, w)$ is computable by induction on the pair $\{u, w\}$ ordered by $(\succ_{horpo})_{lex}$. By property (iii), the neutral term $@(\lambda x.u, w)$ is computable iff v is computable for all v such that $@(\lambda x.u, w) \succ_{horpo} v$. Once more, the definition and Lemma 3.11 imply that the type of v is ground. There are several cases to be considered.

1. If the comparison is by case 5, there are two cases:
   - if $w \succeq_{horpo} v$, we conclude by property (ii).
   - if $\lambda x.u \succ_{horpo} v$, there are two cases. If $u \succeq_{horpo} v$ by case 6, we conclude by property (ii) again. Otherwise, $v = \lambda x.u'$ and $u \succ_{horpo} u'$, implying that u' has a ground type, and $u\{x \mapsto w\} \succ_{horpo} u'\{x \mapsto w\}$ by Lemma 3.6. By assumption and property (ii), $u'\{x \mapsto w\}$ is therefore computable. Hence, $@(v, w)$ which has the same ground type as u' is computable by induction hypothesis applied to the pair $(v = \lambda x.u', w)$. We then conclude by definition of the interpretations that v is computable.
2. If the comparison is by case 9, then $v = @(\overline{v})$ and all terms in $\{\overline{v}\}$ are smaller than w or $\lambda x.u$, hence have a ground type. There are two cases:
   - $v_1 = \lambda x.u$ and $w \succ_{horpo} v_i$ for $i > 1$. Then v_i is computable by property (ii) and, since $u\{x \mapsto v_2\}$ is computable by the main assumption, $@(v_1, v_2)$ is computable by induction hypothesis. If $v = @(v_1, v_2)$, we are done, and we conclude by property (iv) otherwise.
   - For all other cases, terms in $\overline{v}$ are reducts of $\lambda x.u$ and w. Note that reducts of w and reducts of $\lambda x.u$ which are themselves reducts of u are computable by property (ii). Therefore, if all terms in $\overline{v}$ are such reducts, v is computable by property (iv).

   Otherwise, for typing reason, v_1 is a reduct of $\lambda x.u$ of the form $\lambda x.u'$ with $u \succ_{horpo} u'$, and all other terms in $\overline{v}$ are reducts of the previous kind. By the main assumption, $u\{x \mapsto v''\}$ is computable for an arbitrary computable v''. Besides, $u\{x \mapsto v''\} \succ_{horpo} u'\{x \mapsto v''\}$ by Lemma 3.6. Therefore $u'\{x \mapsto v''\}$ is computable for an arbitrary computable v'' by Property (ii). By induction hypothesis, $@(v_1, v_2)$ is again computable. If $v = @(v_1, v_2)$, we are done, otherwise v is computable by property (iv).
3. Otherwise, $@(\lambda x.u, w) \succ_{horpo} v$ by case 11, then $u\{x \mapsto w\} \succeq_{horpo} v$. By assumption, $u\{x \mapsto w\}$ is computable, and hence v is computable by property (ii).

- Property (vi).

  The only if direction is property (i). For the if direction, let s be a strongly normalizable term of ground type $\sigma \in \mathcal{T}^{min}_{\mathcal{S}}$. We prove that s is computable by induction on the definition of $\succ_{horpo}$. Since σ is a data type, s must be neutral. Let now $s \succ_{horpo} t :_C \tau$, hence $\sigma \geq_{\mathcal{T}_{\mathcal{S}}} \tau$ which is therefore ground. By definition of $\mathcal{T}^{min}_{\mathcal{S}}$, $\tau =_{\mathcal{T}_{\mathcal{S}}} \sigma$, hence, by Lemma 3.15, τ is a data type, and since σ is minimal, so is τ, hence $\tau \in \mathcal{T}^{min}_{\mathcal{S}}$. By assumption on s, t must be strongly normalizable, and by induction hypothesis, it is therefore computable. Since this is true of all reducts of s, by definition s is computable.

- Property (vii).

  Assuming that the type σ of $f(\overline{s})$ is ground does not imply, unfortunately, that terms in $\overline{s}$ have a ground type. On the other hand, all other properties have been already proved, hence hold for arbitrary types by the lifting argument. Because we do not need to refer to the definition of the candidate interpretations in the coming proof, but will use instead the proved computability properties, we will be able to carry out the proof without any groundness assumption. In particular, since terms in $\overline{s}$ are computable by assumption, they are strongly normalizable by property (i). We use this remark to build our induction argument: we prove that $f(\overline{s})$ is computable by induction on the pair $(f, \overline{s})$ ordered lexicographically by $(>_{\mathcal{F}}, (\succ_{horpo})_{stat_f})_{lex}$.

  Since $f(\overline{s})$ is neutral, by property (iii), it is computable iff every t such that $f(\overline{s}) \succ_{horpo} t$ is computable, which we prove by an inner induction on the size of t. We discuss according to the possible

cases of the definition of $\succ_{horpo}$.

1. Let $f(\overline{s}) \succ_{horpo} t$ by case 1, hence $s_i \succeq_{horpo} t$ for some $s_i \in \overline{s}$. Since s_i is computable, t is computable by property (ii).
2. Let $s = f(\overline{s}) \succ_{horpo} t$ by case 2. Then $t = g(\overline{t})$, $f >_{\mathcal{F}} g$ and for every $v \in \overline{t}$ either $s \succ_{horpo} v$, in which case v is computable by the inner induction hypothesis, or $u \succeq_{horpo} v$ for some $u \in \overline{s}$ and v is computable by property (ii). Therefore, $\overline{t}$ is computable, and since $f >_{\mathcal{F}} g$, t is computable by the outer induction hypothesis.
3. If $f(\overline{s}) \succ_{horpo} t$ by case 3, then $t = g(\overline{t})$, $f =_{\mathcal{F}} g$, and $\overline{s}(\succ_{horpo})_{mul}\overline{t}$. By definition of the multiset comparison, for every $t_i \in \overline{t}$ there is some $s_j \in \overline{s}$, s.t. $s_j \succeq_{horpo} t_i$, hence, by property (ii), t_i is computable. This allows us to conclude by the outer induction hypothesis that t is computable.
4. If $f(\overline{s}) \succ_{horpo} t$ by case 4, then $t = g(\overline{t})$, $f =_{\mathcal{F}} g$, $\overline{s}(\succ_{horpo})_{lex}\overline{t}$ and for every $v \in \overline{t}$ either $f(\overline{s}) \succ_{horpo} v$ or $u \succeq_{horpo} v$ for some $u \in \overline{s}$. As in the precedence case, this implies that $\overline{t}$ is computable. Then, since $\overline{s}(\succ_{horpo})_{lex}\overline{t}$, t is computable by the outer induction hypothesis.
5. If $f(\overline{s}) \succ_{horpo} t$ by case 7, let $@(t_1, \ldots, t_n)$ be the partial left-flattening of t used in that proof. By the same token as in case 2, every term in $\overline{t}$ is computable, hence t is computable by property (iv).
6. If $f(\overline{s}) \succ_{horpo} t$ by case 8, then $t = \lambda x.u$ with $x \notin \mathcal{V}ar(u)$, and $f(\overline{s}) \succ_{horpo} u$. By the inner induction hypothesis, u is computable. Hence, $u\{x \mapsto w\} = u$ is computable for any computable w, and therefore, $t = \lambda x.u$ is computable by property (v). □

3.5.3 Strong normalization proof

We are now ready for the strong normalization proof.

Lemma 3.18 *Let γ be a type-preserving computable substitution and t be an algebraic λ-term. Then $t\gamma$ is computable.*

Proof: The proof proceeds by induction on the size of t.

1. t is a variable x. Then $x\gamma$ is computable by assumption.
2. t is an abstraction $\lambda x.u$. By Property 3.17 (v), $t\gamma$ is computable if $u\gamma\{x \mapsto w\}$ is computable for every well-typed computable candidate term w. Taking $\delta = \gamma \cup \{x \mapsto w\}$, we have $u\gamma\{x \mapsto w\} = u(\gamma \cup \{x \mapsto w\})$ since x may not occur in γ. Since δ is computable and $|t| > |u|$, by induction hypothesis, $u\delta$ is computable.
3. $t = @(t_1, t_2)$. Then $t_1\gamma$ and $t_2\gamma$ are computable by induction hypothesis, hence t is computable by Property 3.17 (iv).
4. $t = f(t_1, \ldots, t_n)$. Then $t_i\gamma$ is computable by induction hypothesis, hence $t\gamma$ is computable by Property 3.17 (vii). □

We can now easily conclude the proof of well-foundedness needed for our main result, Theorem 3.2, by showing that every term is strongly normalizable with respect to $\succ_{horpo}$.

Proof: Given an arbitrary term t, let γ be the identity substitution. Since γ is type-preserving and computable, $t = t\gamma$ is computable by Lemma 3.18, and strongly normalizable by Property 3.17 (i). □

The restriction of our proof to the first-order sublanguage reduces essentially to Property 3.17 (vii), in which computability is simply identified with strong normalizability. This simple proof of well-foundedness of Dershowitz's recursive path ordering is spelled out in full detail in [27]. The same proof technique had previously been used in a first order context first by Buscholz [10], and later in [19] and [23]. This simple well-foundedness proof of RPO proof does not use Kruskal's tree theorem, and of course, does not show that the recursive path ordering is a well-order. It appears therefore that proving the property of well-foundedness of the recursive path ordering is quite easy while proving the slightly stronger (and useless) property of well-orderedness becomes quite difficult.

3.6 Examples

We now illustrate both the expressive power of HORPO and its weaknesses with more examples. We start with a polymorphic example that shows the power of Case 7 of HORPO.

Example 12 (adapted from [38]) Let $\mathcal{S} = \{\}, \mathcal{S}^\forall = \{\alpha, \beta\}$ and

$$\mathcal{F} = \left\{ \begin{array}{l} nil : List(\alpha), cons : \alpha \times List(\alpha) \Rightarrow List(\alpha), \\ dapply : \beta \times (\beta \to \beta) \times (\beta \to \beta) \Rightarrow \beta, \\ lapply : \beta \times List(\beta \to \beta) \Rightarrow \beta \end{array} \right\}$$

$$\begin{array}{rcrcl} \{F, G : \beta \to \beta,\ x : \beta\} & \vdash & dapply(x, F, G) & \to & F(G(x)) \\ \{x : \alpha\} & \vdash & lapply(x, nil) & \to & x \\ \{F : \beta \to \beta,\ x : \beta,\ l : List(\beta \to \beta)\} & \vdash & lapply(x, cons(F, L)) & \to & F(lapply(x, L)) \end{array}$$

The first rule follows by applying Case 7 twice. The second one holds by Case 1. For the third, we need $List(\alpha) >_{\mathcal{T}_\mathcal{S}} \alpha$ for every type α. Using Case 7, we show that $lapply(x, cons(F, L)) \succ_{horpo} F$ which holds by applying Case 1 twice (note that types decrease) and $lapply(x, cons(F, L)) \succ_{horpo} lapply(x, L)$ which holds by applying Case 3 (taking $lapply \in Mul$) and Case 1 for $cons(F, L) \succ_{horpo} L$. □

The following classical example defines insertion for a polymorphic list. Polymorphism is then exploited by applying the algorithm to particular ascending and descending sorting algorithms for lists of natural numbers.

Example 13 Insertion Sort. Let $\mathcal{S}^\forall = \{\alpha\}$; $\mathcal{S} = \{\mathbb{N} : *, List : * \Rightarrow *\}$

$$\mathcal{F} = \left\{ \begin{array}{l} nil : List(\alpha); cons : \alpha \times List(\alpha) \Rightarrow List(\alpha); \\ insert : \alpha \times List(\alpha) \times (\alpha \to \alpha \to \alpha) \times (\alpha \to \alpha \to \alpha) \Rightarrow List(\alpha); \\ sort : List(\alpha) \times (\alpha \to \alpha \to \alpha) \times (\alpha \to \alpha \to \alpha) \Rightarrow List(\alpha); \\ ascending\_sort, descending\_sort : List(\mathbb{N}) \Rightarrow List(\mathbb{N}); \\ max, min : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N} \end{array} \right\}$$

Let $\Gamma_1 = \{X, Y : \alpha \to \alpha \to \alpha, n : \alpha\}$ and $\Gamma_2 = \{X, Y : \alpha \to \alpha \to \alpha, n, m : \alpha, l : List(\alpha)\}$.

$$\begin{array}{rcrcl} \Gamma_1 & \vdash & insert(n, nil, X, Y) & \to & cons(n, nil) \\ \Gamma_2 & \vdash & insert(n, cons(m, l), X, Y) & \to & cons(X(n, m), insert(Y(n, m), l, X, Y)) \end{array}$$

The first $insert$ rule is easily taken care of by applying Case 2 with the precedence $insert >_\mathcal{F} cons$, and then Case 1. For the second rule, we use Case 2 and recursively need to show two subgoals:

$insert(n, cons(m, l), X, Y) \succ_{horpo} @(X, n, m)$,

which follows by Case 7, and then Case 1 recursively (using $List(\alpha) >_{\mathcal{T}_\mathcal{S}} \alpha$);

$insert(n, cons(m, l), X, Y) \succ_{horpo} insert(Y(n, m), l, X, Y)$,

which follows by Case 4 with a right-to-left lexicographic status for $insert$, calling recursively for the

subgoal $insert(n, cons(m, l), X, Y) \succ_{horpo} @(Y, n, m)$ which is solved by Case 7 and Case 1. We now give the rules for sorting:

$$\begin{array}{rcl} \{X, Y : \alpha \to \alpha \to \alpha\} \vdash sort(nil, X, Y) & \to & nil \\ \{X, Y : \alpha \to \alpha \to \alpha, n : \alpha, l : List(\alpha)\} \vdash sort(cons(n, l), X, Y) & \to & insert(n, sort(l, X, Y), X, Y) \end{array}$$

These rules are easily oriented by $\succ_{horpo}$, with the precedence $sort >_{\mathcal{F}} insert$. We now introduce rules for computing max and min on natural numbers (with either $\{x : \mathbb{N}\}$ or $\{x, y : \mathbb{N}\}$ as environment):

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

We simply need the precedence $max, min >_{\mathcal{F}} s$ for these first-order rules. We come finally to the ascending and descending sort functions for lists of natural numbers, for which typing the righthand sides (with type $\mathbb{N}$) illustrates the use of type substitutions:

$$\begin{array}{rcl} \{l : List(\mathbb{N})\} \vdash ascending\_sort(l) & \to & sort(l, \lambda xy.min(x, y), \lambda xy.max(x, y)) \\ \{l : List(\mathbb{N})\} \vdash descending\_sort(l) & \to & sort(l, \lambda xy.max(x, y), \lambda xy.min(x, y)) \end{array}$$

Unfortunately, $\succ_{horpo}$ fails to orient these two seemingly easy rules. This is so, because the term $\lambda xy.min(x, y)$ occurring in the righthand side has type $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$, which is not comparable to any lefthand side type. We will come back to this example in Section 4. □

We now come to a more tricky example, for which we actually need the transitive closure of the ordering to show termination of a rule, that is, we will need to invent a *middle term* s such that $l \succ_{horpo} s \succ_{horpo} r$ for some rule $l \to r$.

Example 14 (Surjective Disjoint Union, taken from [47]). Let
$\mathcal{S} = \{A, B, U\}$, $\alpha \in \{A, B, U\}$,
$\mathcal{F} = \{inl : A \Rightarrow U; inr : B \Rightarrow U; case_\alpha : U \times (A \to \alpha) \times (B \to \alpha) \Rightarrow \alpha\}$.

$$\begin{array}{rcl} \{X : A, F : A \to \alpha, G : B \to \alpha\} \vdash case_\alpha(inl(X), F, G) & \to & @(F, X) \\ \{Y : B, F : A \to \alpha, G : B \to \alpha\} \vdash case_\alpha(inr(Y), F, G) & \to & @(G, Y) \\ \{Z : U, F : U \to \alpha\} \vdash case_\alpha(Z, \lambda x.H(inl(x)), \lambda y.H(inr(y))) & \to & @(H, Z) \end{array}$$

The type U here is the disjoint union of types A and B, while the $case_\alpha$ function is the associated recursor. This construction exists in functional languages with sum types.

Note that we write $@(F, X)$ instead of $F(X)$ to make clear that $@$ is the top function symbol of the term $F(X)$. For the type ordering we need $A =_{\mathcal{TS}} B =_{\mathcal{TS}} U$. The first two rules are taken care of by Case 7. For the last one, we apply Case 7, and then prove $\lambda x.H(inl(x)) \succ_{horpo} H$ by showing

$$\lambda x.H(inl(x)) \underset{horpo}{\succ} \lambda x.H(x) \underset{horpo}{\succ} H$$

The last comparison holds by Case 12. The first one holds by Cases 10, then Case 9 and finally Case 1.

4 Computability Closure

The ordering is quite sensitive to innocent variations of the rules to be checked, like adding (higher-order) dummy arguments to righthand sides or η-expanding higher-order variables in the righthand sides.

We will solve these problems by improving our definition in the light of the strong normalization proof. In Case 1 of Definition 3.1, we assume that the righthand side term t is indeed smaller or equal to a subterm u of the lefthand side $f(\overline{s})$. Assuming that $\overline{s}$ is computable, we conclude by Property 3.17 (ii) that t is computable. This argument comes again and again in the proofs of the computability properties. Let us now change the definition of Case 1, and require that $u = @(v, w) \succ_{horpo} t$, for some subterms v and w of s. By assumption, v and w are computable, making $@(u, v)$ computable by Property 3.17 (iv), and we are back to the previous case We see here that u may not be a term in $\overline{s}$, but must be built from terms in $\overline{s}$ by computability preserving operations. For example, since a higher-order variable X is computable if and only if $\lambda x.X(x)$ is computable, we may have X as a subterm of the lefthand side of a rule, and $\lambda x.X(x)$ in the righthand side. This discussion is formalized in subsection 4.1 with the notion of a computability closure borrowed from [6], with a slightly enhanced formulation.

4.1 The Computability Closure

Given a set of computable terms, the computability closure builds a superset of computable terms by using computability preserving operations such as those listed in Property 3.17. First, we need a technical definition allowing us to control the type of a selected subterm:

Definition 4.1 *Let $>_{\mathcal{T}_\mathcal{S}}$ be a type ordering. The (strict)* type-decreasing subterm relation*, denoted by $\rhd_{\geq_{\mathcal{T}_\mathcal{S}}}$, is defined as: $s : \sigma \rhd_{\geq_{\mathcal{T}_\mathcal{S}}} t : \tau$ iff $s \rhd t$, $\sigma \geq_{\mathcal{T}_\mathcal{S}} \tau$ and $\mathcal{V}ar(s) \subseteq \mathcal{V}ar(t)$.*

Definition 4.2 *Given a term $t = f(\overline{t})$ with $f \in \mathcal{F}$, we define its* computability 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 typable terms containing all variables in $\mathcal{V}$, all terms in $\overline{t}$, and closed under the following operations:*

1. *subterm of minimal type: let $s \in \mathcal{CC}(t, \mathcal{V})$, and $u : \sigma$ be a subterm of s such that $\sigma \in \mathcal{T}_\mathcal{S}^{min}$ and $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$; then $u \in \mathcal{CC}(t, \mathcal{V})$;*
2. *precedence: let g such that $f >_\mathcal{F} g$, and $\overline{s} \in \mathcal{CC}(t, \mathcal{V})$; then $g(\overline{s}) \in \mathcal{CC}(t, \mathcal{V})$;*
3. *recursive call: let $\overline{s}$ be a sequence of terms in $\mathcal{CC}(t, \mathcal{V})$ such that the term $f(\overline{s})$ is well typed and $\overline{t}(\succ_{horpo} \cup \rhd_{\geq_{\mathcal{T}_\mathcal{S}}})_{stat_f} \overline{s}$; then $g(\overline{s}) \in \mathcal{CC}(t, \mathcal{V})$ for every $g =_\mathcal{F} f$;*
4. *application: let $s : \sigma_1 \to \ldots \to \sigma_n \to \sigma \in \mathcal{CC}(t, \mathcal{V})$ and $u_i : \sigma_i \in \mathcal{CC}(t, \mathcal{V})$ for every $i \in [1..n]$; then $@(s, u_1, \ldots, u_n) \in \mathcal{CC}(t, \mathcal{V})$;*
5. *abstraction: let $x \notin \mathcal{V}ar(t) \cup \mathcal{V}$ and $s \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$; then $\lambda x.s \in \mathcal{CC}(t, \mathcal{V})$;*
6. *reduction: let $u \in \mathcal{CC}(t, \mathcal{V})$, and $u \succeq_{horpo} v$; then $v \in \mathcal{CC}(t, \mathcal{V})$;*
7. *weakening: let $x \notin \mathcal{V}ar(u, t) \cup \mathcal{V}$. Then, $u \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$ iff $u \in \mathcal{CC}(t, \mathcal{V})$.*

As an illustration of the definition, we show that abstraction as well as extensionality are equivalences:

Example 15 Assume that $\lambda x.s \in \mathcal{CC}(t, \mathcal{V})$ and $x \notin \mathcal{V}ar(t) \cup \mathcal{V}$. By weakening, $\lambda x.s \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$. Since $x \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$ by base case of the definition, $@(\lambda x.s, x) \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$ by application case. Therefore, $s \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$ by reduction case. □

Example 16 Assuming that $x \notin \mathcal{V}ar(u) \cup \mathcal{V}$, we show that $\lambda x.@(u,x) \in \mathcal{CC}(t,\mathcal{V})$ iff $u \in \mathcal{CC}(t,\mathcal{V})$.

For the if direction, assuming without loss of generality that $x \notin \mathcal{V}ar(t)$, by weakening, $u \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$. By basic case, $x \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$, hence, by application, $@(u,x) \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$, and by abstraction, we get $\lambda x.@(u,x) \in \mathcal{CC}(t,\mathcal{V})$.

Conversely, assuming that $\lambda x.@(u,x) \in \mathcal{CC}(t,\mathcal{V})$ with $x \notin \mathcal{V}ar(u)$, then $u \in \mathcal{CC}(t,\mathcal{V})$ by reduction, since $\lambda x.@(u,x) \succ_{horpo} u$ by Case 12 of Definition 3.1. □

An important remark is that we use the previously defined ordering $\succeq_{horpo}$ in Case 6 and the relation $\succ_{horpo} \cup \rhd_{\geq_{\mathcal{TS}}}$ in Case 3 of the closure definition instead of simply β-reductions and $\beta \cup \rhd$-reductions respectively as in [32]. And indeed, we will consequently use $\succ_{horpo}$ and $\succ_{horpo} \cup \rhd_{\geq_{\mathcal{TS}}}$ as induction arguments in our proofs. The derivation of the extensionality rule shows the usefulness of rule 6. A price has to be paid for the added expressive power: because $\succ^{+}_{horpo}$ is not order-isomorphic to the natural numbers, the computational closure may be infinite, and since $\succ^{+}_{horpo}$ is probably undecidable, so is the membership of a term to the computability closure. But this membership remains decidable if the number of times Rules 6 and 3 are used is bound. In practice, we can restrict their use by allowing a single step only, which is enough for all examples of the paper. More complex examples to come illustrate how checking membership of a term to a closure can be done by a goal-oriented use of the rules of Defining 4.2.

The following property of the computability closure is shown by induction on the definition:

Lemma 4.3 *Assume that $u \in \mathcal{CC}(t)$. Then, $u\gamma \in \mathcal{CC}(t\gamma)$ for every type-preserving substitution γ.*

Proof: We prove that if $u \in \mathcal{CC}(t,\mathcal{V})$ with $\mathcal{V} \subseteq \mathcal{X} \setminus (\mathcal{V}ar(t) \cup \mathcal{V}ar(t\gamma) \cup \mathcal{D}om(\gamma))$, then $u\gamma \in \mathcal{CC}(t\gamma,\mathcal{V})$. We proceed by induction on the definition of $\mathcal{CC}(t,\mathcal{V})$. Note that the property on $\mathcal{V}$ depends on t and γ, but not on u. It will therefore be trivially satisfied in all cases but abstraction and weakening. And indeed, these are the only cases in the proof which are not routine, hence we do them in detail as well as Case 1 to show its simplicity.

Case 1: let $s = f(\overline{s}) : \sigma \in \mathcal{CC}(t,\mathcal{V})$ with $f \in \mathcal{F}$ and $u : \tau$ be a subterm of s with $\sigma \in \mathcal{T}^{min}_{\mathcal{S}}$ and $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$. By induction hypothesis, $s\gamma \in \mathcal{CC}(t\gamma,\mathcal{V})$. By assumption on u, $u\gamma : \tau$ is a subterm of $s\gamma : \sigma$ and $\mathcal{V}ar(u\gamma) \subseteq \mathcal{V}ar(t\gamma)$. Therefore, $u\gamma \in \mathcal{CC}(t\gamma,\mathcal{V})$ by Case 1.

Case 5: let $u = \lambda x.s$ with $x \in \mathcal{X} \setminus (\mathcal{V} \cup \mathcal{V}ar(t))$ and $s \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$. To the price of renaming the variable x in s if necessary, we can assume in addition that $x \notin \mathcal{V}ar(t\gamma) \cup \mathcal{D}om(\gamma)$, and therefore $\mathcal{V} \cup \{x\} \subseteq \mathcal{X} \setminus (\mathcal{V}ar(t) \cup \mathcal{V}ar(t\gamma) \cup \mathcal{D}om(\gamma))$. By induction hypothesis, $s\gamma \in \mathcal{CC}(t\gamma, \mathcal{V} \cup \{x\})$. Since $x \notin \mathcal{V}ar(t\gamma) \cup \mathcal{V}$, by Case 5 of the definition $\lambda x.s\gamma \in \mathcal{CC}(t\gamma,\mathcal{V})$ and, since $x \notin \mathcal{D}om(\gamma)$, we have $u\gamma = \lambda x.s\gamma$.

Case 7: let $\mathcal{V}' = \mathcal{V} \cup \{x\}$. By definition, $u \in \mathcal{CC}(t,\mathcal{V}')$, with $x \notin \mathcal{V}ar(u,t) \cup \mathcal{V}$. By induction hypothesis, $u\gamma \in \mathcal{CC}(t\gamma,\mathcal{V}')$. Since $x \notin \mathcal{V}ar(u\gamma, t\gamma) \cup \mathcal{V}$, $u\gamma \in \mathcal{CC}(t\gamma,\mathcal{V})$ by Case 7.

□

Lemma 4.4 *Assume that $u : \sigma \in \mathcal{CC}(t : \tau)$. Then, $u\delta : \sigma\xi_\delta \in \mathcal{CC}(t\delta : \tau\xi_\delta)$ for every specialization δ.*

Proof: The proof is similar as previously, but uses the polymorphic property of $\succeq_{horpo}$ in Case 6.

4.2 The Higher-Order Recursive Path Ordering with Closure

From now on, both orderings $\succ_{horpo}$ and $\succ_{chorpo}$ will coexist. They differ only by the definition of the property A, and by the first, subterm case. As a way to stress their similarities, we do not reformulate the entire ordering. Environments and types are again omitted here.

$$A = \forall v \in \bar{t}\ s \underset{chorpo}{\succ} v \text{ or } u \underset{chorpo}{\succeq} v \text{ for some } u \in \mathcal{CC}(s)$$

Definition 4.5

$$s : \sigma \underset{chorpo}{\succ} t : \tau \textit{ iff } \sigma \geq_{\mathcal{T}_S} \tau \textit{ and}$$

1. $s = f(\bar{s})$ *with* $f \in \mathcal{F}$, *and (i)* $u \underset{chorpo}{\succeq} t$ *for some* $u \in \bar{s}$ *or (ii)* $t \in \mathcal{CC}(s)$

Cases 2 to 12 are kept unchanged, $\succ_{horpo}$ *(resp.* $\succeq_{horpo}$*) being replaced by* $\succ_{chorpo}$ *(resp.* $\succeq_{chorpo}$*).*

The definition is recursive, since recursive calls operate on pairs of terms which decrease in the well-founded ordering $(\succ_{horpo}, \rhd)_{right-to-left-lex}$.

As an easy consequence of Case 1, we obtain that $s = f(\bar{s}) : \sigma \succ_{chorpo} t : \tau$ with $f \in \mathcal{F}$ if $\sigma = \tau$ and $t \in \mathcal{CC}(s)$. This shows that the technique based on the general schema, essentially based on the closure mechanism introduced by Blanqui, Jouannaud and Okada [6], is a very particular case of the present method. Besides, this new method inherits all the advantages of the recursive path ordering, in particular it is possible to combine it with interpretation based techniques [12].

The proofs of Lemmas 3.8, 3.6 and 3.7 are easy to adapt (using now Lemma 4.3 for the proof of the new version of Lemma 3.6 and Lemma 4.4 for Lemma 3.7) to the ordering with closure:

Lemma 4.6 $\succ_{chorpo}$ *is monotonic, stable, and polymorphic.*

4.3 Strong Normalization

We first show that terms in the computability closure of a term are computable.

First, the computability predicate is of course now defined with respect to $\succ_{chorpo}$. To show that the computability properties remain valid, we simply observe that there is no change whatsoever in the proofs, since the rules applying to application-headed terms did not change. For this, it is crucial to respect the given formulation of Case 5, avoiding the use of the closure as in Case 1. Actually, we could have defined a specific, weaker closure for terms headed by an application, to the price of doing again the two most complex proofs of the computability properties. We did not think it was worth the trouble.

Second, we are still using the previous ordering $\succ_{horpo}$ in the definition of the new ordering $\succ_{chorpo}$, via the closure definition in particular. We therefore need to prove that $\succ_{horpo} \subseteq \succ_{chorpo}$ in order to apply the induction arguments based on the well-foundedness of $\succ_{chorpo}$ on some appropriate set of terms:

Lemma 4.7 $\succ_{horpo} \subseteq \succ_{chorpo}$.

The proof is easily done by induction since all cases in the definition of $\succ_{horpo}$ appear in the definition of $\succ_{chorpo}$. The importance of this lemma comes from the computability properties. By the above inclusion, any term smaller than a computable term in the ordering $\succ_{horpo}$ will therefore be computable by Property 3.17 (ii).

We may of course wonder whether $\succ_{horpo}^{+} \subseteq \succ_{chorpo}$. Let $s_1 \succ_{horpo} s_2 \succ_{horpo} \ldots \succ_{horpo} s_n$. Then $s_1 \succ_{chorpo} s_n$ in case $s_n \in \mathcal{CC}(s_1)$. This is in particular true when $s_2 \in \mathcal{CC}(s_1)$, since Case 6 of the closure definition then implies that $s_n \in \mathcal{CC}(s_1)$. This is not true in general since $s_1 \notin \mathcal{CC}(s_1)$.

We now show that $\succ_{horpo}$, which is monotonic for terms of equivalent types and well-founded, remains well-founded when combined with $\rhd_{\geq_{\mathcal{T}_\mathcal{S}}}$:

Lemma 4.8 *Let $>$ be a well-founded relation on terms which is monotonic for terms of equivalent types (in $=_{\mathcal{T}_\mathcal{S}}$). Then, $> \cup \rhd_{\geq_{\mathcal{T}_\mathcal{S}}}$ is well-founded.*

Proof: Since $>$ is well-founded, it is enough to show the absence of infinite decreasing sequences for terms of equivalent types, which follows from the fact that $\rhd_{\geq_{\mathcal{T}_\mathcal{S}}}$ and $>$ commute for such terms by monotonicity of $>$. □

We now come to the main property of terms in the computability closure, which justifies its name:

Property 4.9 *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 $(>_\mathcal{F}, (\succ_{horpo} \cup \rhd_{\geq_{\mathcal{T}_\mathcal{S}}})_{stat_f})_{lex}$ operating on pairs $\langle f, \overline{t} \rangle$. Then every term in $\mathcal{CC}(t)$ is computable.*

The precise formulation of this statement arises from its forthcoming use inside the proof of Lemma 4.10.

Proof: We prove that $u\gamma : \sigma$ 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 $u \in \mathcal{V}$, then $u\gamma$ is computable by the assumption on γ; if $u \in \overline{t}$, we conclude by the assumption that $\overline{t}$ is computable, since $u\gamma = u$ by the assumption that $\mathcal{V} \cap \mathcal{V}ar(t) = \emptyset$; and if u is a subterm of $f(\overline{t})$, we again remark that γ acts as the identity on such terms, and therefore, they are computable by assumption.

For the induction step, we discuss the successive operations to form the closure:

case 1: u is a subterm of type in $\mathcal{T}_\mathcal{S}^{min}$ of some $v \in \mathcal{CC}(t, \mathcal{V})$. By induction hypothesis, $v\gamma$ is computable, hence strongly normalizable by Property 3.17 (i). By monotonicity of $\succ_{chorpo}$, its subterm $u\gamma$ is also strongly normalizable, and hence computable by Property 3.17 (vi).

case 2: $u = g(\overline{u})$ where $\overline{u} \in \mathcal{CC}(t, \mathcal{V})$. By induction hypothesis, $\overline{u}\gamma$ is computable. Since $f >_\mathcal{F} g$, $u\gamma$ is computable by our assumption that terms smaller than $f(\overline{t})$ are computable.

case 3: $u = g(\overline{u})$ where $f =_\mathcal{F} g$, $\overline{u} \in \mathcal{CC}(t, \mathcal{V})$ and $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$. By induction hypothesis, $\overline{u}\gamma$ is computable. By assumption, and stability of the ordering under substitutions, $\overline{t}\gamma (\succ_{horpo} \cup \rhd_{\geq_{\mathcal{T}_\mathcal{S}}})_{stat_f} \overline{u}\gamma$. Note that $\overline{t}\gamma = \overline{t}$ by our assumption that $\mathcal{V} \cap \mathcal{V}ar(t) = \emptyset$. Therefore $u\gamma = g(\overline{u}\gamma)$ is computable by our assumption that terms smaller than $f(\overline{t})$ are computable.

case 4: by induction and Property 3.17 (iv).

case 5: let $u = \lambda x.s$ with $x \notin \mathcal{V}$ 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 $x \notin \mathcal{D}om(\gamma) \cup \mathcal{V}ar(t)$. As a first consequence, $\mathcal{V} \cup \{x\} \cap \mathcal{V}ar(t) = \emptyset$; as a second, 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 3.17 (v), $(\lambda x.s)\gamma$ is computable.

case 6: by induction, stability and Property 3.17 (ii).

case 7: by induction hypothesis. □

We now restate Property 3.17 (vii) and show in one case how the proof makes use of Property 4.9.

Lemma 4.10 *Let $f \in \mathcal{F}$ and let $\bar{t}$ be a set of terms. If $\bar{t}$ is computable, then $f(\bar{t})$ is computable.*

Proof: We prove that $f(\bar{t})$ is computable if terms in $\bar{t}$ are computable by an outer induction on the pair $\langle f, \bar{t}\rangle$ ordered lexicographically by the ordering $(>_{\mathcal{F}}, (\succ_{chorpo} \cup \rhd_{\geq_{\mathcal{TS}}})_{stat_f})_{lex}$, and an inner induction on the size of the reducts of t. Since terms in $\bar{t}$ are computable by assumption, they are strongly normalizable by Property 3.17(ii), hence, by Lemma 4.8 the ordering $(>_{\mathcal{F}}, ((\succ_{chorpo} \cup \rhd_{\geq_{\mathcal{TS}}})_{stat})_{lex}$ is well founded on the pairs satisfying the assumptions.

By Lemma 4.7, the set of pairs smaller than the pair $\langle f, \bar{t}\rangle$ for the ordering $(>_{\mathcal{F}}, (\succ_{chorpo} \cup \rhd_{\geq_{\mathcal{TS}}})_{stat})_{lex}$ contains the set of pairs that are smaller than that pair for the ordering $(>_{\mathcal{F}}, (\succ_{horpo} \cup \rhd_{\geq_{\mathcal{TS}}})_{stat})_{lex}$. This key remark allows us to use Property 4.9.

The proof is actually similar to the proof of Property 3.17 (vii), except for Case 1 and for the cases using property A. We therefore do Cases 1 and 2, the latter using property A.

case 1: let $f(\bar{t}) \succ_{chorpo} s$ by case 1, hence $t_i \succeq_{horpo} s$ for some $t_i \in \bar{t}$ or $s \in \mathcal{CC}(t)$. In the former case, since t_i is computable, s is computable by Property 3.17 (ii); in the latter case, all terms smaller than $f(\bar{t})$ with respect to $(>_{\mathcal{F}}, (\succeq_{horpo})_{stat})_{lex}$ are computable by induction hypothesis and the above key remark, hence s is computable by Property 4.9.

case 2: let $t = f(\bar{t}) \succ_{chorpo} s$ by case 2. Then $s = g(\bar{s})$, $f >_{\mathcal{F}} g$ and for every $s_i \in \bar{s}$ either $t \succ_{chorpo} s_i$, in which case s_i is computable by the inner induction hypothesis, or $v \succeq_{chorpo} s_i$ for some $v \in \mathcal{CC}(t)$, in which case v is computable by Property 4.9 and hence s_i is computable by Property 3.17 (ii). We then conclude that s is computable the by outer induction hypothesis since $f >_{\mathcal{F}} g$. □

Theorem 4.11 *$\succ^{+}_{chorpo}$ is a polymorphic higher-order reduction ordering.*

Proof: Thanks to Property 4.9, the strong normalization proof of this improved ordering is exactly the same as the one for HORPO, using of course Lemma 4.10 instead of Property 3.17 (vii). Using now Lemma 4.6, we therefore conclude that the transitive closure of $\succ_{chorpo}$ is a higher-order polymorphic reduction ordering. □

4.4 Examples

This new definition of the ordering is much stronger than the previous one. In addition to allowing us proving the strong normalization property of the remaining rules of the sorting example, it also allows proving termination of the following rule, which is added to the rules of Example 8:

Example 17 $x * y \;\; \rightarrow \;\; rec(y, 0, \lambda z_1 z_2.x + z_2)$

This rule can be proved terminating with the precedence $* >_{\mathcal{F}} \{rec, +, 0\}$. Indeed,
$x * y \succ_{chorpo} rec(y, 0, \lambda z_1 z_2.x + z_2)$ by Case 2 of $\succ_{chorpo}$, since $x * y \succ_{chorpo} y$ by case 1, $x * y \succ_{chorpo} 0$ by case 2 again, and $\lambda z_1 z_2.x + z_2 \in \mathcal{CC}(x * y)$:
applying Case 5 of the definition of the computability closure twice, we need to show $x + z_2 \in \mathcal{CC}(x * y, \{z_1, z_2\})$. By Case 2 we are left with $x \in \mathcal{CC}(x * y, \{z_1, z_2\})$ and $z_2 \in \mathcal{CC}(x * y, \{z_1, z_2\})$, which both hold by base Case of the definition of the computability closure. □

The coming example is quite classical too.

Example 18 Let

$$\mathcal{S} = \{List : * \Rightarrow *\}; \mathcal{S}^{\forall} = \{\alpha\}$$
$$\mathcal{F} = \left\{ \begin{array}{l} 0, 1 : \alpha; + : \alpha \times \alpha \Rightarrow \alpha; nil : List(\alpha); cons : \alpha \times List(\alpha) \Rightarrow List(\alpha); \\ foldl : (\alpha \to \alpha \to \alpha) \times \alpha \times List(\alpha) \Rightarrow \alpha; sum : List(\alpha) \Rightarrow \alpha; +^c : \alpha \to \alpha \Rightarrow \alpha \end{array} \right\}$$

$$\begin{array}{rcrcl} \{x : \alpha,\ F : \alpha \to \alpha \to \alpha\} & \vdash & foldl(F, x, nil) & \to & x \\ \{x, y : \alpha,\ F : \alpha \to \alpha \to \alpha,\ l : List(\alpha)\} & \vdash & foldl(F, x, cons(y, l)) & \to & foldl(F, (F\ x\ y), l) \\ \{\} & \vdash & +^c & \to & \lambda xy.x + y \\ \{l : List(\alpha)\} & \vdash & sum(l) & \to & foldl(+^c, 0, l) \end{array}$$

The first rule is by subterm case.

For the second, we set a right-to-left lexicographic status for $foldl$, and, applying Case 4, we recursively have to show that (i) $F \succeq_{chorpo} F$; (ii) $foldl(F, x, cons(y, l)) \succ_{chorpo} @(F, x, y)$, which succeeds easily by rule 7; and (iii) $cons(y, l) \succ_{chorpo} l$, which succeeds by subterm case provided $List(\alpha) \geq_{\mathcal{T}_S} \alpha$.

For the third, we show that $\lambda xy.x + y$ is in the closure of $+^c$, provided $+^c >_{\mathcal{F}} +$, by successively applying Case 5 twice, Case 2 and, finally, the base Case twice.

For the last, we add the precedence $sum >_{\mathcal{F}} \{foldl, +^c, 0\}$ in order to prove it by Case 2 of $\succ_{chorpo}$, followed by Case 2 for $sum(l) \succeq_{chorpo} 0$ and showing that $+^c \in \mathcal{CC}(sum(l))$, by Case 2. □

The following example, a definition of formal derivation, illustrates best the power of the computability closure.

Example 19 Let $\mathcal{S} = \{\Re\}$ be the sort of real numbers.

$$\mathcal{F} = \left\{ \begin{array}{l} D : (\Re \to \Re) \Rightarrow (\Re \to \Re); \\ 0, 1 :\to \Re;\ -, sin, cos, ln : \Re \Rightarrow \Re; \\ +, \times, / : \Re \times \Re \Rightarrow \Re \end{array} \right\}$$

$$\begin{array}{rcrcl} & & D(\lambda x.y) & \to & \lambda x.0 \\ & & (\lambda x.x) & \to & \lambda x.1 \\ & & D(\lambda x.sin(x)) & \to & \lambda x.cos(x) \\ & & D(\lambda x.cos(x)) & \to & \lambda x. - sin(x) \\ \{F, G : \Re \to \Re\} & \vdash & D(\lambda x.(F\ x) + (G\ x)) & \to & \lambda x.(D(F)\ x) + (D(G)\ x) \\ \{F, G : \Re \to \Re\} & \vdash & D(\lambda x.(F\ x) \times (G\ x)) & \to & \lambda x.(D(F)\ x) \times (G\ x) + (F\ x) \times (D(G)\ x) \\ \{F : \Re \to \Re\} & \vdash & D(\lambda x.ln(F\ x)) & \to & \lambda x.(D(F)\ x)/(F\ x) \end{array}$$

We take $D >_{\mathcal{F}} \{0, 1, \times, -, +, /\}$ for precedence and assume that the function symbols are in Mul. Apart from the first rule, this example makes a heavy use of the closure. The computations involved are quite complex, and can be followed by using our implementation. The but-last rule, in particular, is entirely processed by the closure mechanism (together with Case 1). This does not mean that it could be already solved with the same proof by the technique developed in [6], where the computability closure was first introduced without any other mechanism. The point is that the closure defined here is more powerful thanks to case 6 based on $\succ_{horpo}$ instead of simply using β-reductions as in [6].

We are not completely satisfied with this computation, though, because the structural definition of the ordering has been completely lost here. We would like to improve the ordering so as to do more computations with the ordering and delegate the hard parts only to the more complex closure mechanism. For the time being, however, the closure mechanism is the only one which applies to rules whose righthand

side is an abstraction with the bound variable occurring in the body, assuming that the lefthand side of rule is not itself an abstraction. We discuss its implementation in Section 5.2.

The next example (currying/uncurrying) shows again the use of a middle term.

Example 20 First, to a signature $\mathcal{F}$, we associate the signature

$$\mathcal{F}^{curry} = \left\{ \begin{array}{l} f_i : \sigma_1 \times \ldots \times \sigma_i \Rightarrow \sigma_{i+1} \rightarrow \ldots \rightarrow \sigma_p \rightarrow \tau \text{ for every } i \in [0..p] \mid \\ f : \sigma_1 \times \ldots \times \sigma_n \Rightarrow \sigma_{n+1} \rightarrow \ldots \rightarrow \sigma_p \rightarrow \tau \in \mathcal{F} \text{ where } \tau \text{ is a data-type} \end{array} \right\}$$

and we introduce the following sets of rewrite rules for currying/uncurrying:

$$\begin{array}{lrcl} j \in J \subseteq [1..p]: & \{t_1 : \sigma_1, \ldots, t_{j+1} : \sigma_{j+1}\} \vdash & f_{j+1}(t_1, \ldots, t_{j+1}) & \rightarrow \quad @(f_j(t_1, \ldots, t_j), t_{j+1}) \\ i \in I \subseteq [1..p]: & \{t_1 : \sigma_1, \ldots, t_{i+1} : \sigma_{i+1}\} \vdash & @(f_i(t_1, \ldots, t_i), t_{i+1}) & \rightarrow \quad f_{i+1}(t_1, \ldots, t_{i+1}) \end{array}$$

When $i = j$, the above rules are clearly non-terminating since they originate from the same equation oriented in both directions. Termination therefore requires that the two subsets I and J of $[0..p]$ are disjoint. We postpone their precise definition.

Starting with the first rule, we set $f_{j+1} >_{\mathcal{F}} f_j$. Applying case 7 we have to show that $f_j(t_1, \ldots, t_j) \in \mathcal{CC}(f_{j+1}(t_1, \ldots, t_{j+1}))$ which holds by the precedence case followed by the base case for all the arguments.

Moving now to the second rule, we set $f_i >_{\mathcal{F}} f_{i+1}$. The only possible case is subterm for application, that is Case 5. But the obtained subgoal $f_i(t_1, \ldots, t_i) \succ_{chorpo} f_{i+1}(t_1, \ldots, t_{i+1})$ cannot succeed since there is a new free variable (t_{i+1}) in the righthand side. The computability closure does not help either here, since it is defined for terms headed by an algebraic function symbol.

The trick is to invent a middle term, and show that the lefthand side is bigger than the righthand one in the *transitive closure* of the ordering. The convenient middle term here is $@(\lambda x.f_{i+1}(t_1, \ldots, t_i, x), t_{i+1})$, which reduces to the righthand side by the use of Case 11. We therefore simply need to show that the lefthand side is bigger than the middle term. Since both terms are headed by an abstraction, by Case 9 we have to prove that $f_i(t_1, \ldots, t_i) \succ_{chorpo} \lambda x.f_{i+1}(t_1, \ldots, t_i, x)$, which we prove now by showing that the righthand side term is in the closure of the lefthand side one: by abstraction case we need $f_{i+1}(t_1, \ldots, t_i, x) \in \mathcal{CC}(f_i(t_1, \ldots, t_i), \{x\})$, and by precedence case, we are left with $t_1, \ldots t_i, x \in \mathcal{CC}(f_i(t_1, \ldots, t_i), \{x\})$, which holds by the base case.

Although the use of a middle terms looks like a lucky trick, the conditions for applying it successfully can be easily characterized and hence implemented, making it transparent for the user. This is described in Section 5.

We can accommodate a mixture of currying and uncurrying by choosing an appropriate well-founded precedence for selecting the right number of arguments desired for a given function symbol: let $J = [1..m]$ and $I = [m + 1..p]$ for some $m \in [1..p]$, and let us add the rule

$$\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \vdash f(x_1, \ldots, x_n) \rightarrow f_n(x_1, \ldots, x_n)$$

which is easily proved terminating by adding $f >_{\mathcal{F}} f_n$ to the precedence. The resulting set of rules will then replace all occurrences of f by f_m while adding or eliminating the necessary application operators.

4.5 A mutually inductive definition of the ordering and the computability closure

So far, we restrained ourselves using $\succ_{chorpo}$ in the computability closure definition, and used instead $\succ_{horpo}$ in order to avoid a mutually inductive definition of the ordering and the closure. This allowed

us to present both separately, starting with the computability closure which is built on a simple intuitive idea. Using $\succ_{chorpo}$ recursively in Cases 3 and 6 of the closure definition would of course yield a stronger relation. In this section, we show that this indeed yields a well-founded ordering.

First, we need to show that the mutually recursive definition yields a leat fixpoint. This is the case since the underlying functional is defined by a set of Horn clauses, hence is monotone.

Then, we need to show that the computability properties remain true when using this new definition. Indeed, there is no impact on any of the proofs but thde proof of Lemma 4.9. However, it suffices to replace the assumption based on the ordering $(>_{\mathcal{F}}, (\succ_{horpo} \cup \rhd_{\geq_{\mathcal{T}_\mathcal{S}}})_{stat_f})_{lex}$ by the very same assumption based now on the stronger ordering $(>_{\mathcal{F}}, (\succ_{chorpo} \cup \rhd_{\geq_{\mathcal{T}_\mathcal{S}}})_{stat_f})_{lex}$. The proof then goes exactly as before.

Finally, we remark that the proof of Lemma 4.10 is itself based on an induction with the ordering $(>_{\mathcal{F}}, (\succ_{chorpo} \cup \rhd_{\geq_{\mathcal{T}_\mathcal{S}}})_{stat_f})_{lex}$, hence can be kept as it is. Note however that we needed to prove that $\succ_{horpo}$ was included into $\succ_{chorpo}$. This was Lemma 4.7, needed to relate both inductions, with respect to $(>_{\mathcal{F}}, (\succ_{horpo} \cup \rhd_{\geq_{\mathcal{T}_\mathcal{S}}})_{stat_f})_{lex}$ in Lemma 4.9, and with respect to $(>_{\mathcal{F}}, (\succ_{chorpo} \cup \rhd_{\geq_{\mathcal{T}_\mathcal{S}}})_{stat_f})_{lex}$ in Lemma 4.10. This is no more needed, hence the need for lemma 4.7 disappears.

5 Implementation

A PROLOG implementation is available from our web page whose principles are described here. We define first an ordering on types satisfying the required properties before explaining how to approximate $(\succ_{horpo})^*$ and the closure mechanism.

5.1 The recursive path ordering on types

Given a partial quasi-ordering $\geq_{\mathcal{S}}$ on sort constructors again called a *precedence*, as well as a status, we define the following rpo-like quasi ordering on types:

$$\text{Let } B = \forall v \in \overline{\tau}\ \sigma >_{\mathcal{T}_\mathcal{S}} v$$

Definition 5.1 *$\sigma \geq_{\mathcal{T}_\mathcal{S}} \tau$ iff*

1. *$\sigma = c(\overline{\sigma})$ for some $c \in \mathcal{S}$ and $\sigma_i \geq_{\mathcal{T}_\mathcal{S}} \tau$ for some $\sigma_i \in \overline{\sigma}$*
2. *$\sigma = c(\overline{\sigma})$ and $\tau = d(\overline{\tau})$ for some $c, d \in \mathcal{S}$ such that $c >_{\mathcal{S}} d$, and B*
3. *$\sigma = c(\overline{\sigma})$ and $\tau = d(\overline{\tau})$ for some $c, d \in \mathcal{S}$ such that $c =_{\mathcal{S}} d$ and $\overline{\sigma}(\geq_{\mathcal{T}_\mathcal{S}})_{mul}\overline{\tau}$*
4. *$\sigma = c(\overline{\sigma})$ and $\tau = d(\overline{\tau})$ for some $c, d \in \mathcal{S}$ such that $c =_{\mathcal{S}} d$ and $\overline{\sigma}(\geq_{\mathcal{T}_\mathcal{S}})_{lex}\overline{\tau}$, and B*
5. *$\sigma = \alpha \to \beta$, and $\beta \geq_{\mathcal{T}_\mathcal{S}} \tau$*
6. *$\sigma = \alpha \to \beta$, $\tau = \alpha' \to \beta'$, $\alpha =_{\mathcal{T}_\mathcal{S}} \alpha'$ and $\beta \geq_{\mathcal{T}_\mathcal{S}} \beta'$*

Note that, because of the subterm property for sort symbols (since they belong to a first-order unisorted structure), our above definition of B ($\forall v \in \overline{\tau}\ \sigma >_{\mathcal{T}_\mathcal{S}} v$) is equivalent to the former definition of A applied to types ($\forall v \in \overline{\tau}\ \sigma >_{\mathcal{T}_\mathcal{S}} v$ or $u \geq_{\mathcal{T}_\mathcal{S}} v$ for some $u \in \overline{\sigma}$). This remark will be used in Section 6.3 to build a single uniform ordering operating on both terms and types.

Proposition 5.2 *$\geq_{\mathcal{T}_\mathcal{S}}$ is a type ordering and $>_{\mathcal{T}_\mathcal{S}}^{\rightarrow}$ is the recursive path ordering generated by the given precedence on $\mathcal{S}$.*

Proof: It is clear that adding the subterm property for the arrow to the definition of $>_{\mathcal{T}_\mathcal{S}}$ yields the recursive path ordering (note that the precedence is not changed, hence symbols in $\mathcal{S}$ do not compare with the arrow). This shows the latter property and implies therefore well-foundedness of $>_{\mathcal{T}_\mathcal{S}}$. Arrow-preservation and arrow-decreasingness are built-in. Stability is proved by induction on types. □

All examples given so far can use the above type ordering, provided the appropriate precedence on type constructors is given by the user as well as their status.

5.2 Implementable approximation of $\succ_{horpo}$ and $\succ_{chorpo}$

Among the many issues that must be dealt with, we consider here only the *new* ones. Guessing a precedence and a status for the function symbols and the type constructors is by no means different from what it is for Dershowitz's recursive path ordering, and the technology is therefore well-known. Accordingly, this is not done by our current implementation: the user is supposed to input the precedence to the system which can then *check* a set of rules. On the other hand, guessing a middle term or how to implement the closure mechanism effectively are new implementation problems that we consider here.

Property A**.** The non-deterministic or comparison of proposition A used in cases 2, 4, and 7, can be replaced by the equivalent deterministic one:

$$\forall v : \rho \in \bar{t} \text{ if } \rho \leq_{\mathcal{T}_\mathcal{S}} \tau \ \ then\ s \underset{horpo}{\succ} v \text{ otherwise } u \underset{horpo}{\succeq} v \text{ for some } u : \theta \in \bar{s} \text{ such that } \theta \geq_{\mathcal{T}_\mathcal{S}} \rho$$

Left-flattening. There is a tradeoff between the increase of types and the decrease of size when using left-flattening in Case 9: moving from $@(@(a,b),c))$ to $@(a,b,c)$ replaces the subterm $@(a,b)$ by the two smaller subterms a, b, but a has a bigger type than $@(a,b)$. Type considerations may therefore be used to drive the choice of the amount of flattening.

Guessing middle terms. A weakness of our definition is that the relation $\succ_{horpo}$ does not satisfy transitivity. This is why our statement in Theorem 3.2 uses $(\succ_{horpo})^*$. In most examples, we have used $\succ_{horpo}$, and indeed, the lack of transitivity has two origins: left-flattening and Cases 11 and 12. Therefore, the use of these two cases will usually come together with the need of guessing a middle term u such that $s \succ_{horpo} u \succ_{horpo} t$, the second comparison being done by one of the above two cases. It turns out that it is quite easy to guess when such a middle term may be necessary: when no case other than 11 and 12 applies to compare s and t. More specifically, when we need to compare a term s headed by an application with a term t headed by a function symbol. In this case, a middle term u can be generated by β-expanding t, and we can now try applying Case 9 between s and u and Case 11 between u and t. Similarly, in case s is an abstraction whose λ cannot be pulled out, we can η-expand t before comparing s and u by Case 10 and then u and t by Case 12.

Approximation of the closure. Our implementation of the closure mechanism is an approximation as well, by weakening proposition A defined in Section 4.2 so as to be the same as in the subterm case (Case 1), that is $A = \forall v \in \vec{t}\ s \succ_{chorpo} v$ or $v \in \mathcal{CC}(s)$, which amounts to have a simple guess of the term u defined in the afore mentioned version of A to be t itself. Reduction is only applied to the arguments of the starting term itself, it can therefore now be seen as a new basic case. The implementation of the other rules defining the closure is goal-directed. We have implemented a subset of them, namely application, abstraction, precedence, recursive call and reduction which turned out to suffice for all our

examples -we of course had proved these examples by hand before, but the resulting proofs were indeed more complex proofs and used a larger set of rules. In the current version of the implementation, we use $\succ_{chorpo}$ recursively in the computability closure definition, rather than $\succ_{horpo}$ so as to enhance the expressivity of the ordering.

Examples All examples considered in the paper have been checked with our implementation, therefore supporting our claim that HORPO is automatable. The only user inputs are the following: the type constructors, their arity and their precedence; the function symbols, their type and their precedence; the terms to be compared and the type of their free variables. The implementation checks the type of both terms to be compared in the environment formed by the type of the variables, and proceeds with checking the ordering with the approximation of $\succ_{chorpo}$ that we just described.

6 From Past to Future Work

6.1 Comparisons

Higher-order termination is a difficult problem with several instances.

Higher-order rules using plain pattern matching are routinely used in proof systems with inductive types, at the object level (Gödel's recursor is one example), and at the type level as well (under the name of strong elimination in the Calculus of Inductive Constructions [14]). These rules have a specific format, hence their strong normalization property can be proved by ad'hoc arguments. By allowing rule-based definitions in proof systems, one provides with more general induction schemas than the usual structural induction schema for inductive types. This however requires user-defined plain higher-order rules with the recursor rules for inductive types as particular cases. Then, showing that proof-checking is decidable, and the obtained proof system is sound, requires proving that these rules are strongly normalizing (together with the built-in $\beta\eta$ rules). Our program aims at proving the strong normalization property of such rules, therefore including the recursor rules for inductive types, and provide a generic tool for that purpose. So far, we have answered this question within the framework of polymorphic higher-order algebras only. Our future target is a framework including dependent types, therefore containing the Calculus of Inductive Constructions as a special case.

Higher-order rewrite rules are also used in logical systems to describe computations over lambda-terms used as a suitable abstract syntax for encoding functional objects like programs or specifications. This approach has been pioneered in this context by Nipkow [40] and is available in Isabelle [44]. Its main feature is the use of higher-order pattern matching for firing rules. A recent generalization of Nipkow's setting allows one for rewrite rules of polymorphic, higher-order type [30, 26]. Proving the strong normalization property of such rules is a different problem, somewhat harder, since requiring a well-founded ordering compatible with $\beta\eta$-equivalence classes of terms, and it is well known that building compatible well-founded orderings is a hard problem as exemplified with the case of associativity and commutativity in the first-order case.

There is still a third way of defining higher-order rewriting, due to Klop, which came actually first [36]. Retrospectively, this approach appears to be a sort of instance of Nipkow's more recent approach, in the sense that, given a set of rules, a typed term rewrites in Klop sense when it does so in Nipkow's sense. Since Klop was more interested in confluence than in termination properties, the question of proving termination of Klop's rewrite relation has not been investigated.

Surprisingly, the question of designing orderings for proving higher-order termination attracted attention first within Nipkow's framework. The initial attempts, however, including ours, were not very

convincing. All methods were based either on very weak orderings [31], or required interaction with a general purpose higher-order theorem prover [47]. Only recently were we able to answer the question with a decidable ordering able to prove all examples we have found in the literature [33]. What is interesting to note is that this ordering builds on the present one in a decisive way.

Let us therefore now compare the different attempts to prove termination of plain higher-order rules.

First, it should be noted that the present definition is by far superior to the one given in [32], which could only compare terms of the same type, or of equivalent types. Consequently, we are able to solve many more examples. Besides, the definition of the ordering has been enriched considerably, by adding new cases. We are convinced that Definition 3.1 cannot be substantially improved, except by: handling explicitly bound variables in the recursive calls; using the computability properties of subterms in presence of inductive types; using interpretations as it was successfully done for the recursive path ordering [34] and for the first version of the higher-order recursive path ordering [12]. On the other hand, we hope that the more complex closure mechanism can be improved and integrated to the ordering. These improvements are carried out in part in [8].

There is an alternative to using the higher-order recursive path ordering for proving termination of plain higher-order rewriting, based on the computability closure and the general schema [7]. Because the main construction in [5] is the computability closure, it can indeed be obtained by restricting $\succ_{chorpo}$ to its first case, and is therefore much weaker for our framework of polymorphic algebras. On the other hand, this method is more advanced insofar it allows for dependent types and even for rules, like strong elimination, operating at the type level. This lets us hope that generalizing the higher-order recursive path ordering to dependent types may be at reach.

Let us finally mention two completely different methods by Jones and Bohr [25] and by Giesl, Thiemann and Schneider-Kamp [21]. They differ from ours by targeting functional programs rather than proof systems, and encoding these programs as a set of termination-preserving applicative rules obtained by Currying [35]. Termination of these applicative rules is then analyzed by various methods. However, since the structure of terms is lost because of the translation, these methods must be based on an analysis of the flow of redexes in terms. This analysis must of course cope with the flow of beta-redexes via the translation. In presence of polymorphic or dependent types, there is no method known, apart from Girard's, which is able to account for the flow of redexes in lambda-terms. Therefore, we do not think that these method can help us solving our grand challenge. In the simpler case of functional programs, these methods may be quite effective whenever termination does not depend upon the type structure used. We believe that the orderings methods considered here are inherently more powerful since they can in principle easily integrate semantic information such as that obtained from an analysis of the flow of redexes.

6.2 Future work

The higher-order recursive path ordering should be seen as a firm step to undergo further developments in different directions. Two of them have been investigated in the first order framework: the case of associative commutative operators, and the use of interpretations as a sort of elaborated precedence operating on function symbols. The second extension has been carried out for a restricted higher-order framework [12]. Both deserve immediate attention. Other extensions are specific to the higher-order case: incorporating the computability closure into the ordering definition; enriching the type system with inductive types, a problem considered in the framework of the general schema by Blanqui [7, 5]; enriching the type system with dependent types, considered for the original version of the higher-order recursive path ordering [32] by Walukiewicz [50]. We suggest below a path to an extension of the

higher-order recursive path ordering defined here to a dependent type framework.

6.3 A uniform ordering on terms and their types

We are going here to give a single definition working uniformly on terms and types. Its restriction to types will be the type ordering defined in Section 3, while its restriction to terms will be the higher-order recursive path ordering using this type ordering. This ensures that the new definition is equivalent to Definition 3.1 when using the type ordering introduced in Section 5, since terms and types do not interfere in our framework. The definition is very uniform, working by induction, as it is usual for the recursive path ordering, with cases based on the top operator of the lefthand and righthand expressions to be compared. What makes this uniform definition possible is that the type ordering is a restriction of Dershowitz's recursive path ordering for first-order terms, and that the higher-order recursive path ordering restricts to Dershowitz's recursive path ordering when comparing first-order terms.

For making uniformity possible, we add a new constant in our language, $*$, such that all types have themselves type $*$. We omit the straightforward type system for typing types, which only aims at verifying arities of sorts symbols. $*$ will therefore be the only non-typable term in the language.

6.3.1 Statuses and Precedence

We assume given

- a partition $Mul \uplus Lex$ of $\mathcal{F} \cup \mathcal{S}$;
- a well-founded precedence $\geq_{\mathcal{FS}}$ on $\mathcal{F} \cup \mathcal{S}$ such that $\geq_{\mathcal{FS}}$ is the union of a quasi-ordering $\geq_{\mathcal{F}}$ on $\mathcal{F}$ and a quasi-ordering $\geq_{\mathcal{T_S}}$ on $\mathcal{S}$ whose strict parts are well-founded; variables are considered as constants incomparable in $>_{\mathcal{FS}}$ among themselves and with other function symbols.

6.3.2 Definition of the ordering

Definition 6.1

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

$$\textit{Given } s : \sigma \textit{ and } t : \tau, \quad s \underset{horpo}{\succ} t \textit{ iff } \sigma = \tau = * \textit{ or } \sigma \underset{horpo}{\succeq} \tau \textit{ and}$$

1. $s = f(\overline{s})$ *with* $f \in \mathcal{FS}$, *and* $u \underset{horpo}{\succeq} t$ *for some* $u \in \overline{s}$
2. $s = f(\overline{s})$ *and* $t = g(\overline{t})$ *with* $f >_{\mathcal{FS}} g$, *and* A
3. $s = f(\overline{s})$ *and* $t = g(\overline{t})$ *with* $f =_{\mathcal{FS}} g \in Mul$ *and* $\overline{s}(\underset{horpo}{\succ})_{mul}\overline{t}$
4. $s = f(\overline{s})$ *and* $t = g(\overline{t})$ *with* $f =_{\mathcal{FS}} g \in Lex$ *and* $\overline{s}(\underset{horpo}{\succ})_{lex}\overline{t}$, *and* A
5. $s = @(s_1, s_2)$ *and* $s_1 \underset{horpo}{\succeq} t$ *or* $s_2 \underset{horpo}{\succeq} t$
6. $s = \lambda x : \sigma.u$, $x \notin \mathcal{V}ar(t)$ *and* $u \underset{horpo}{\succeq} t$
7. $s = f(\overline{s})$, $@(\overline{t})$ *is an arbitrary left-flattening of* t, *and* A

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

9. $s = @(s_1, s_2)$, $@(\overline{t})$ *is an arbitrary left-flattening of* t *and* $\{s_1, s_2\}(\underset{horpo}{\succ})_{mul}\overline{t}$

10. $s = \lambda x : \alpha.u,\ t = \lambda x : \beta.v$, $\alpha{=}\beta$ *and* $u \underset{horpo}{\succ} v$

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

12. $s = \lambda x.@(u, x)$ *with* $x \notin \mathcal{V}ar(u)$ *and* $u \underset{horpo}{\succeq} t$

13. $s = \alpha \to \beta$, *and* $\beta \underset{horpo}{\succeq} t$

14. $s = \alpha \to \beta,\ t = \alpha' \to \beta'$, $\alpha \underset{horpo}{=} \alpha'$ *and* $\beta \underset{horpo}{\succ} \beta'$

This uniform higher-order recursive path ordering operating on terms and types opens the way to its generalization to dependent type calculi such as the Calculus of Constructions.

7 Conclusion

We have defined a powerful mechanism for defining polymorphic reduction orderings on higher-order terms that include β- and η-reductions. To the best of our knowledge, this is the first such ordering ever. Moreover, these orderings can be implemented without much effort, as witnessed by our own prototype implementation and the many examples proved. And because these orderings restrict to Dershowitz's recursive path ordering on first-order terms, a hidden achievement of our work is a new, simple, easy to teach well-foundedness proof for Dershowitz's recursive path ordering itself.

Our construction includes the computable closure mechanism originating from [6], where it was used to define a syntactic class of higher-order rewrite rules that are compatible with beta reductions and with recursors for arbitrary positive inductive types. The language there is indeed richer than the one considered here, since it is the calculus of inductive constructions generated by a monomorphic signature. The usefulness of the notion of closure in these different context shows the strength of the concept.

Using the computability technique instead of the Kruskal theorem to prove the strong normalization property of the ordering is intriguing, and raises the question whether it is possible to exhibit a suitable extension of Kruskal's theorem that would allow proving that the higher-order recursive path ordering is a well-order of the set of higher-order terms ? Here, we must confess that we actually failed in our initial quest to find one. In retrospect, the reason is that we were looking for too strong a statement. It may be that a version of Kruskal's theorem holds based on an adequate notion of subterm such as the weak subterm property satisfied by HORPO when dropping Case 11. We do not think that any Kruskal-like theorem holds when Case 11 is included.

Acknowledgments: we are grateful to Frédéric Blanqui, Adam Koprowski and Femke Van Ramsdoonk for several useful remarks and to a careful referee for scrutinizing the paper and making important suggestions.

References

[1] Franz Baader and Tobias Nipkow. Term Rewriting and All That. Cambridge University Press, 1999.

[2] Henk Barendregt. *Handbook of Theoretical Computer Science*, volume B, chapter Functional Programming and Lambda Calculus. North-Holland, 1990. J. van Leeuwen ed.

[3] Henk Barendregt. *Handbook of Logic in Computer Science*, chapter Typed lambda calculi. Oxford Univ. Press, 1993. Abramsky et al. eds.

[4] M. Bezem, J.W. Klop and R. de Vrijer eds. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science 55, Cambridge University Press, 2003.

[5] F. Blanqui. Inductive Types in the Calculus of Constructions. In TLCA, *Lecture Notes in Computer Science* 2701:395–409. Springer-Verlag, 2003.

[6] F. Blanqui, J.-P. Jouannaud, and M. Okada. The Calculus of Algebraic Constructions. In RTA, *Lecture Notes in Computer Science* 1631:301–316. Springer-Verlag, 1999.

[7] F. Blanqui, J.-P. Jouannaud, and M. Okada. Inductive Data Types. *Theoretical Computer Science* 277:41–68, 2002.

[8] F. Blanqui, J.-P. Jouannaud and A. Rubio. Higher-Order Termination: from Kruskal to Computability. In LPAR, *Lecture Notes in Computer Science* 4246:1–14. Springer-Verlag, 2006.

[9] F. Blanqui, J.-P. Jouannaud, and P.-Y. Strub. The Calculus of Congruent Constructions. draft available from the web at http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud.

[10] W. Buchholz. Proof-theoretic analysis of termination proofs. *Annals Pure Applied Logic* 75(1-2):57–65, 1995.

[11] C. Borralleras, M. Ferreira and A. Rubio. Complete monotonic semantic path orderings. In CADE, *Lecture Notes in Artificial Intelligence* 1831:346–364. Springer-Verlag, 2000.

[12] C. Borralleras and A. Rubio. A monotonic, higher-order semantic path ordering. In LPAR, *Lecture Notes in Computer Science* 2250:531–547. Springer-Verlag, 2001.

[13] A. Colmerauer. Equations and Inequations on Finite and Infinite Trees. In FGCS, pp. 85–99. OHMSHA Ltd. Tokyo and North-Holland, 1984.

[14] T. Coquand and C. Paulin-Mohring. Inductively defined types. In COLOG-88, *Lecture Notes in Computer Science* 417:50–66. Springer-Verlag, 1990.

[15] T. Coquand. Inductive definitions and type theory. In Second Int. summer school in logic for computer science. Chambéry, France, 1994.

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

[17] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*, volume B:243–309. North-Holland, 1990.

[18] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, 2003.

[19] M. Fernández and J.-P. Jouannaud. Modular termination of term rewriting systems revisited. In *Recent Trends in Data Type Specification, Lecture Notes in Computer Science* 906:255–272. Springer-Verlag, 1995.

[20] J. Gallier. On Girard's "Candidats de Reductibilité". In Logic and Computer Science, pp. 123–203. Academic Press, 1990.

[21] J. Giesl, R. Thiemann, P. Schneider-Kamp. Proving and Disproving Termination of Higher-Order Functions. In FroCoS, *Lecture Notes in Artificial Intelligence* 3717:216–231, Springer Verlag, 2005.

[22] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[23] J. Goubault-Larrecq. Well founded recursive relations. In CSL 2001, *Lecture Notes in Computer Science* 2142:484–497. Springer-Verlag, 2002.

[24] G. Huet, G. Kahn and C. Paulin-Mohring. The Coq Proof Assistant. A Tutorial. Version 7.3. INRIA Rocquencourt and ENS Lyon.

[25] N. Jones and N. Bohr. Termination Analysis of the Untyped lambda-Calculus. In RTA, *Lecture Notes in Computer Science* 3091:1–23. Springer-Verlag, 2004.

[26] J.P. Jouannaud. Higher-order rewriting: framework, confluence and termination. In *Processes, Terms and Cycles: Steps on the road to infinity*. Essays Dedicated to Jan Willem Klop on the occasion of his 60th Birthday. Springer Verlag, 2005.

[27] J.-P. Jouannaud. Extension orderings revisited. Available from the web at http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud.

[28] J-P. Jouannaud and M. Okada. A Computation Model for Executable Higher-Order Algebraic Specifications Languages. In LICS, pp. 350–361. IEEE Computer Society Press, 1991.

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

[30] J-P. Jouannaud and F. van Raamsdonk and A. Rubio. Higher-order Rewriting with Types and Arities. Available from the web at http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud.

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

[32] J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In LICS, pp. 402–411. IEEE Computer Society Press, 1999.

[33] J.-P. Jouannaud and A. Rubio. Higher-Order Orderings for Normal Rewriting. In RTA, *Lecture Notes in Computer Science* 4098:387–399. Springer-Verlag, 2006.

[34] S. Kamin and J.-J. Levy. Two generalizations of the recursive path ordering. Technical report, Department of computer science, University of Illinois at Urbana-Champaign, 1980.

[35] J.R. Kennaway, J.W. Klop, M.R. Sleep, and F.J. de Vries. Comparing curried and uncurried rewriting. *Journal of Symbolic Computation*, 21:15–39, 1996.

[36] J. W. Klop. Combinatory Reduction Relations. Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.

[37] J. W. Klop. Term Rewriting Systems. In *Handbook of Logic in Computer Science*, volume 2:2–116. Oxford University Press, 1992.

[38] C. Loría-Sáenz and J. Steinbach. Termination of combined (rewrite and λ-calculus) systems. In CTRS, *Lecture Notes in Computer Science* 656:143–147. Springer-Verlag, 1992.

[39] A. Mal'cev. On the elementary theories of locally free algebras. *Soviet Math. Doklady*, 1961.

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

[41] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[42] T. Nipkow. Higher-order critical pairs. In LICS, pp. 342–349. IEEE Computer Society Press, 1991.

[43] M. Okada and G. Takeuti. On the theory of quasi ordinal diagrams. In Logic and Combinatorics, 1985. *Contemporary Mathematics* 65:295–308. American Mathematical Society, 1987.

[44] L. C. Paulson. Isabelle: the next 700 theorem provers. In *Logic and Computer Science*, pp. 361–386. Academic Press, 1990.

[45] J. van de Pol. Termination proofs for higher-order rewrite systems. In HOA 1993, *Lecture Notes in Computer Science* 816:305–325. Springer-Verlag, 1994.

[46] J. van de Pol. *Termination of Higher-Order Rewrite Systems*. PhD thesis, Utrecht University, The Netherlands, 1996.

[47] J. van de Pol and H. Schwichtenberg. Strict functional for termination proofs. In TLCA, *Lecture Notes in Computer Science* 902:350-364. Springer-Verlag, 1995.

[48] F. van Raamsdonk. Confluence and Normalization for Higher-Order Rewrite Systems. Phd thesis, Vrije Universiteit, Amsterdam, The Netherlands, 1996.

[49] A. Rubio. A fully syntactic AC-RPO. In RTA, *Lecture Notes in Computer Science* 1631:133-147, Springer-Verlag, 1999.

[50] D. Walukiewicz-Chrzaszcz. Termination of rewriting in the Calculus of Constructions. *Journal Functional Programming*, 13(2):339–414, 2003.