HORPO with Computability Closure : A Reconstruction

Frédéric Blanqui[1], Jean-Pierre Jouannaud[2⋆], and Albert Rubio[3]

[1] INRIA & LORIA, Protheo team, Campus Scientifique, BP 239, 54506 Vandœuvre-lès-Nancy Cedex, France
[2] LIX, École Polytechnique, 91400 Palaiseau, France
[3] Technical University of Catalonia, Pau Gargallo 5, 08028 Barcelona, Spain

Abstract. This paper provides a new, decidable definition of the higher-order recursive path ordering in which type comparisons are made only when needed, therefore eliminating the need for the computability closure, and bound variables are handled explicitly, making it possible to handle recursors for arbitrary strictly positive inductive types.

1 Introduction

The Higher-order Recursive Path ordering was first introduced in [3]. The goal was to provide a tool for showing strong normalization of simply typed lambda calculi in which higher-order constants were defined by higher-order recursive rules using plain pattern matching. Inspired by Dershowitz's recursive path ordering for first-order terms, comparing two terms started by comparing their types under a given congruence generated by equating given basic types, before to proceed recursively on the structure of the compared terms. In [4], the type discipline was generalized to a polymorphic type discipline with type constructors, the congruence on types was replaced by a well-founded quasi-ordering on types (in practice, a restriction of the recursive path ordering on types), and the recursive definition itself could handle new cases. There were two variants of the subterm case: in the first, following the recursive path ordering tradition, a subterm of the left-hand side was compared with the whole right-hand side; in the second, a term belonging to the computability closure of the left-hand side was used instead of a subterm. And indeed, a subterm is the basic case of the computability closure construction, whose fixpoint definition included various operations under which Tait and Girard's notion of computability is closed. The ordering and the computational closure definitions shared a lot in common, raising some expectations for a simpler and yet more expressive definition

⋆ Project LogiCal, Pôle Commun de Recherche en Informatique du Plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

able to handle inductive types, as advocated in [2]. This paper meets these expectations (and goes indeed much further) with a new definition of HORPO that improves over the previous one [4] in several respects:

1. There is a single decidable recursive definition, instead of a pair of mutually inductive definitions for the computability closure and the ordering itself;
2. In contrast with the definition of HORPO with computability closure, the new definition is decidable and syntax-directed (except, as usual, for the subterm case);
3. Type checking applies only when really needed, that is, when the comparison does not follow from computability arguments;
4. Bound variables are handled explicitly by the ordering, allowing for arbitrary abstractions in the right-hand sides;
5. Strictly positive inductive types are accommodated;
6. There is no need for flattening applications on the right-hand side.

This new definition appears to be powerful enough to prove strong normalization of recursors for arbitrary strictly positive inductive types. The two major technical innovations which make it possible are the integration of the computability closure within the ordering definition on the one hand, and the explicit handling of binders on the other hand. This integration of the computability closure is not obtained by adding new cases in the definition, as was suggested in [2], but instead by eliminating from the previous definition the unnecessary type checks.

2 Higher-Order Algebras

Polymorphic higher-order algebras are introduced in [4]. Their purpose is twofold: to define a simple framework in which many-sorted algebra and typed lambda-calculus coexist; to allow for polymorphic types for both algebraic constants and lambda-calculus expressions. For the sake of simplicity, we will restrict ourselves to monomorphic types in this presentation, but allow us for polymorphic examples. Carrying out the polymorphic case is no more difficult, but surely more painful.

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

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

Types are *functional* when headed by the $\rightarrow$ symbol, and *data types* otherwise. $\rightarrow$ associates to the right. We use $\sigma, \tau, \rho, \theta$ for arbitrary types.

Function symbols are meant to be algebraic operators equipped with a fixed number n of arguments (called the *arity*) of respective types $\sigma_1, \ldots, \sigma_n$, and an *output type* σ. Let $\mathcal{F} = \biguplus_{\sigma_1,\ldots,\sigma_n,\sigma} \mathcal{F}_{\sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma}$. The membership of a given function symbol f to $\mathcal{F}_{\sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma}$ is called a *type declaration* and written $f : \sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma$.

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:

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

The raw term $\lambda x : \sigma.u$ is an *abstraction* and $@(u, v)$ is an application. We may omit σ in $\lambda x : \sigma.u$ and write $@(u, v_1, \ldots, v_n)$ or $u(v_1, \ldots, v_n)$, $n > 0$, omitting applications. $\mathcal{V}ar(t)$ is the set of free variables of t. A raw term t is *ground* if $\mathcal{V}ar(t) = \emptyset$. The notation $\overline{s}$ shall be ambiguously used for a list, a multiset, or a set of raw terms $s_1, \ldots, s_n$.

Raw terms are identified with finite labeled trees by considering $\lambda x : \sigma.u$, for each variable x and type σ, as a unary function symbol taking u as argument to construct the raw term $\lambda x : \sigma.u$. *Positions* are strings of positive integers. $t|_p$ denotes the *subterm* of t at position p. We use $t \trianglerighteq t|_p$ for the subterm relationship. The result of replacing $t|_p$ at position p in t by u is written $t[u]_p$.

An *environment* Γ is a finite set of pairs written as $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$, where x_i is a variable, σ_i is a type, and $x_i \neq x_j$ for $i \neq j$. Our typing judgements are written as $\Gamma \vdash_{\Sigma} s : \sigma$. A raw term s has type σ in the environment Γ if the judgement $\Gamma \vdash_{\Sigma} s : \sigma$ is provable in the inference system given at Figure 1. An important property of our type system is that a raw term typable in a given environment has a unique type. Typable raw terms are called *terms*. We categorize terms into three disjoint classes:

1. *Abstractions* headed by λ;
2. *Prealgebraic* terms headed by a function symbol, assuming that the output type of $f \in \mathcal{F}$ is a base type;
3. *Neutral* terms are variables or headed by an application.

A *substitution* σ of domain $\mathcal{D}om(\sigma) = \{x_1, \ldots, x_n\}$ is a set of triples $\sigma = \{\Gamma_1 \vdash_{\Sigma} x_1 \mapsto t_1, \ldots, \Gamma_n \vdash_{\Sigma} x_n \mapsto t_n\}$, such that x_i and t_i have the same type in the environment Γ_i. Substitutions are extended to terms by morphism, variable capture being avoided by renaming bound variables when necessary. We use post-fixed notation for substitution application.

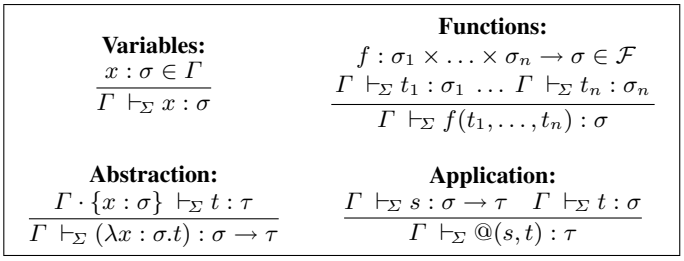

Fig. 1. The type system for monomorphic higher-order algebras

A rewrite rule is a triple $\Gamma \vdash_\Sigma l \to r$ such that $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$, and $\Gamma \vdash_\Sigma l : \sigma$ and $\Gamma \vdash_\Sigma r : \sigma$ for some type σ. Given a set of rules R, for example the beta- and eta- rules of the lambda-calculus,

$$s \xrightarrow[l \to r \in R]{p} t \text{ iff } s|_p = l\gamma \text{ and } t = s[r\gamma]_p \text{ for some substitution } \gamma$$

The notation $l \to r \in R$ assumes that the variables bound in l, r (resp. the variables free in l, r) are renamed away from the free variables of $s[]_p$ (resp. the bound variables of $s[]_p$), to avoid captures.

For simplicity, typing environments are omitted in the rest of the paper.

A *higher-order reduction ordering* $\succ$ is a well-founded ordering of the set of typable terms which is

(i) *monotonic*: $s \succ t$ implies that $u[s] \succ u[t]$;

(ii) *stable*: $s \succ t$ implies that $s\gamma \succ t\gamma$ for all substitution γ.

(iii) *functional*: $s \longrightarrow_\beta \cup \longrightarrow_\eta t$ implies $s \succ t$,

In [4], we show that the rewrite relation generated by $R \cup \{\text{beta,eta}\}$ can be proved by simply checking that $l > r$ for all $l \to r \in R$ with some higher-order reduction ordering.

3 The Improved Higher-Order Recursive Path Ordering

The improved higher-order recursive path ordering on higher-order terms is generated from four basic ingredients: a *type ordering*; an *accessibility* relationship; a *precedence* on functions symbols; and a *status* for the function symbols. Accessibility is a new ingredient originating in inductive types, while the other three were already needed for defining HORPO. We describe these ingredients before defining the improved higher-order recursive path ordering.

3.1 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 $\tau \to \alpha \geq_{\mathcal{T}_\mathcal{S}} \sigma \to \alpha$;

  We denote by $\mathcal{T}_\mathcal{S}^{min}$ the set of minimal types with respect to $>_{\mathcal{T}_\mathcal{S}}^{\to} = >_{\mathcal{T}_\mathcal{S}} \cup \rhd$.

  We say that a data type σ occurs *positively* (resp. *negatively*) in a type τ if τ is a data type (resp. τ is a data type non equivalent to σ in $=_{\mathcal{T}_\mathcal{S}}$), or if $\tau = \rho \to \theta$ and σ occurs positively (resp. negatively) in θ and negatively (resp. positively) in ρ.
- a set $Acc(f)$ of accessible arguments for each function declaration $f : \sigma_1 \ldots \sigma_n \to \sigma$ such that σ is a data type : $i \in [1..n]$ is said to be *accessible* if all data types occuring in σ_i are smaller than σ in the quasi-order $\geq_{\mathcal{T}_\mathcal{S}}$, and in case of equivalence (with $=_{\mathcal{T}_\mathcal{S}}$), they must occur positively in σ_i. Note that the application operator $@ : (\alpha \to \beta) \times \alpha \to \beta$ can be seen as a function symbol with an empty set of accessible positions, since its output type τ may occur negatively in any of its two argument types σ and $\sigma \to \tau$.

  A term u is *accessible* in $f(\overline{s})$, $f \in \mathcal{F}$, iff $u = s_i$ or u is *accessible* in s_i for some $i \in Acc(f)$. *Accessibility* for $f \in \mathcal{F} \cup \{@\}$ is now obtained by adding the minimal type subterms: $s = f(\overline{s}) \rhd_{acc} v : \tau$ iff v is accessible in $\overline{s}$, or $\tau \in \mathcal{T}_\mathcal{S}^{min}$ and $\mathcal{V}ar(v) \subseteq \mathcal{V}ar(s)$.
- a *precedence* $\geq_\mathcal{F}$ on symbols in $\mathcal{F} \cup \{@\}$, with $f >_\mathcal{F} @$ for all $f \in \mathcal{F}$.
- a status for symbols in $\mathcal{F} \cup \{@\}$ with $@ \in Mul$.

We recall important properties of the type ordering [4]:

Lemma 1. *Assuming $\sigma =_{\mathcal{T}_\mathcal{S}} \tau$, σ is a data type iff τ is a data type.*

Lemma 2. *Let $\geq_{\mathcal{T}_\mathcal{S}}$ be a quasi-ordering on types such that $>_{\mathcal{T}_\mathcal{S}}$ is well-founded, arrow monotonic and arrow preserving. Then, the relation $\geq_{\mathcal{T}_\mathcal{S}}^{\to} = (\geq_{\mathcal{T}_\mathcal{S}} \cup \rhd)^*$ is a well-founded quasi-ordering on types extending $\geq_{\mathcal{T}_\mathcal{S}}$ and $\rhd$, whose equivalence coincides with $=_{\mathcal{T}_\mathcal{S}}$.*

Lemma 3. *$\mathcal{T}_\mathcal{S}^{min}$ is a non-empty set of data types if $\mathcal{T}_\mathcal{S} \neq \emptyset$.*

3.2 Notations

- $s \succ^X t$ for the main ordering, with a finite set of variables $X \subset \mathcal{X}$, with the convention that X is omitted when empty;
- $s : \sigma \succ^X_{\mathcal{T}_\mathcal{S}} t : \tau$ for $s \succ^X t$ and $\sigma \geq_{\mathcal{T}_\mathcal{S}} \tau$;
- $s \trianglerighteq_{acc} \succeq^X_{\mathcal{T}_\mathcal{S}} t$ for $s \trianglerighteq_{acc} w$ for some w and $@(w, \overline{x}) : \sigma' =_{\mathcal{T}_\mathcal{S}} \sigma \succeq_{\mathcal{T}_\mathcal{S}} t$ for some $\overline{x} \in X$.

3.3 Ordering definition

Definition 1. $s : \sigma \succ^X t : \tau$ *iff either:*

1. $s = f(\overline{s})$ *with* $f \in \mathcal{F}$ *and either of*
   (a) $s_i \trianglerighteq_{acc} \succeq^X_{\mathcal{T}_\mathcal{S}} t$ *for some* i
   (b) $t = g(\overline{t})$ *with* $f =_\mathcal{F} g \in \mathcal{F}$, $s \succ^X \overline{t}$ *and* $\overline{s}(\succ_{\mathcal{T}_\mathcal{S}} \cup \trianglerighteq_{acc} \succeq^X_{\mathcal{T}_\mathcal{S}})_{stat_f} \overline{t}$
   (c) $t = g(\overline{t})$ *with* $f >_\mathcal{F} g \in \mathcal{F} \cup \{@\}$ *and* $s \succ^X \overline{t}$
2. $s = @(u, v)$ *and either of*
   (a) $u \trianglerighteq_{acc} \succeq^X_{\mathcal{T}_\mathcal{S}} t$ *or* $v \trianglerighteq_{acc} \succeq^X_{\mathcal{T}_\mathcal{S}} t$
   (b) $t = @(u', v')$ *and* $\{u, v\}(\succ^X_{\mathcal{T}_\mathcal{S}})_{mul}\{u', v'\}$
   (c) $u = \lambda x : \alpha.w$ *and* $w\{x \mapsto v\} \succeq^X t$
3. $s = \lambda x : \alpha.u$ *and either of*
   (a) $u\{x \mapsto z\} \succeq^X_{\mathcal{T}_\mathcal{S}} t$ *for* $z : \alpha$ *fresh*
   (b) $t = \lambda y : \beta.v$, $\alpha =_{\mathcal{T}_\mathcal{S}} \beta$ *and* $u\{x \mapsto z\} \succ^X v\{y \mapsto z\}$ *for* $z : \beta$ *fresh*
   (c) $u = @(v, x)$, $x \notin \mathcal{V}ar(v)$ *and* $v \succeq^X t$
4. (a) $s \notin \mathcal{X}$ *and* $t \in X$
   (b) $s \notin \mathcal{X}, s \neq \lambda x : \alpha.u$, $t = \lambda y : \beta.w$ *and* $s \succ^{X \cup \{z\}} w\{y \mapsto z\}$ *for* $z : \beta$ *fresh*

Our ordering definition comes in four parts, dealing the first three with left-hand sides headed respectively by an algebraic symbol, the application symbol and an abstraction, while the fourth factors out those cases where the right-hand side is a previously bound variable or an abstraction.

Cases 1 are very similar (up to type checks) to those of Dershowitz's recursive path ordering with the subterm case 1a, the status case 1b and the precedence case 1c. So are Cases 2 and 3. One difference is that there is an additional case for handling respectively beta and eta. A more substantial difference is that variable renaming has become explicit.

The major innovation of this new definition is the annotation of the ordering by the set of variables X that were originally bound in the right-hand side term, but have become free by taking some subterm. This allows rule 4b to pull out abstractions from the right-hand side regardless

of the left-hand side term, meaning that abstractions are smallest in the precedence. Note that freed variables become smaller than everything else but variables.

One may wonder why Case 1b is so complicated: the reason is that using recursively $\overline{s}(\succ_{\mathcal{T}_\mathcal{S}}^{X})_{stat_f}\overline{t}$ would yield non-termination.

We give now an example of use of this new definition with the inductive type of Brouwer's ordinals, whose constructor lim takes an infinite sequence of ordinals to build a new, limit ordinal, hence admits a functional argument of type $\mathbb{N} \to Ord$, in which Ord occurs positively. As a consequence, the recursor admits a much more complex structure than that of natural numbers, with an explicit abstraction in the right-hand side of the rule for lim:

$rec(lim(F), U, X, W) \to @(W, F, \lambda n.rec(@(F, n), U, X, W))$

Although the strong normalization of such rules is known to be difficult to prove, it is checked automatically by our ordering:

Example 1. Brouwer's ordinals.

$0 : Ord \qquad S : Ord \Rightarrow Ord \qquad lim : (\mathbb{N} \to Ord) \Rightarrow Ord$

$n : \mathbb{N} \qquad F : \mathbb{N} \to Ord$

$rec : Ord \times \alpha \times (Ord \to \alpha \to \alpha) \times ((\mathbb{N} \to Ord) \to (\mathbb{N} \to \alpha) \to \alpha) \Rightarrow \alpha$

1. $rec(lim(F), U, X, W) \succ_{\mathcal{T}_\mathcal{S}} @(W, F, \lambda n.rec(@(F, n), U, X, W))$ yields 4 subgoals according to Case 1c:
2. $\alpha \geq_{\mathcal{T}_\mathcal{S}} \alpha$ which is trivially satisfied, and
3. $rec(lim(F), U, X, W) \succ \{W, F, \lambda n.rec(@(F, n), U, X, W)\}$ which simplifies to:
4. $rec(lim(F), U, X, W) \succ W$ which succeeds by Case 1a,
5. $rec(lim(F), U, X, W) \succ F$, which succeeds by Case 1a since F is accessible in $lim(F)$,
6. $rec(lim(F), U, X, W) \succ \lambda n.rec(@(F, n), U, X, W)$ which yields by Case 4b
7. $rec(lim(F), U, X, W) \succ^{\{n\}} rec(@(F, n), U, X, W)$ which yields by Case 1b
8. $\{lim(F), U, X, W\}(\succ_{\mathcal{T}_\mathcal{S}} \cup \rhd_{acc} \succeq_{\mathcal{T}_\mathcal{S}}^{\{n\}})_{mul}\{@(F, n), U, X, W\}$, which reduces to
9. $lim(F) \rhd_{acc} \succeq_{\mathcal{T}_\mathcal{S}}^{\{n\}} @(F, n)$ which succeeds by Case 1a since F is accessible in $lim(F)$,
10. $rec(lim(F), U, X, W) \succ^{\{n\}} \{@(F, n), U, X, W\}$, our remaining goal, decomposes into three goals trivially solved by Case 1a, that is
11. $rec(lim(F), U, X, W) \succ^{\{n\}} \{U, X, W\}$, and one additional goal

12. $rec(lim(F), U, X, W) \succ^{\{n\}} @(F, n)$ which yields two goals by Case 1c
13. $rec(lim(F), U, X, W) \succ^{\{n\}} F$, which succeeds by Case1a, since F is accessible in $lim(F)$, and
14. $rec(lim(F), U, X, W) \succ^{\{n\}} n$ which succeeds by Case 4a, therefore ending the computation.

4 Strong normalization

Theorem 1. *$(\succ_{\mathcal{T}_S})^+$ is a decidable higher-order reduction ordering.*

Contrasting with our previous proposal made of an ordering part and a computability closure part, our new ordering is a decidable inductive definition: $s \succ^X t$ is defined by induction on the triple (n, s, t), using the order $(>_{\mathbb{N}}, \longrightarrow_\beta \cup \rhd, \rhd)_{lex}$, where n is the number of abstractions in t. The quadratic time decidability follows since all operations used are clearly decidable in linear time. The fact that $\succ^X$ is quadratic comes from those cases that recursively compare one side with each subterm of the other side. This assumes of course that precedence and statuses are given, since inferring them yields NP-completeness as is well-known for the recursive path ordering on first-order terms.

The stability and monotonicity proofs are routine. As the old one, the new definition is not transitive, but this is now essentially due to the beta-reduction case 2c. We are left with strong normalization, and proceed as in [4]. The computability predicate differs however in case of data types, since it has to care about inductive type definitions.

4.1 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 $=_{\mathcal{T}_S}$, a set of terms $[\![\sigma]\!]$ closed under term formation. 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, which may arise in case t does not have type τ but $\tau' =_{\mathcal{T}_S} \tau$. Relaxing the type system to type terms up to type equivalence $=_{\mathcal{T}_S}$ is routine [4]. We use the notation $t :_C \sigma$ to indicate that the raw term t, called a *candidate term* (or simply, a term), has type σ in the relaxed system.

4.2 Candidate interpretations

In the coming sections, we consider the well-foundedness of the strict ordering $(\succ_{\mathcal{T}_S})^+$, that is, equivalently, the strong normalization of the

rewrite relation defined by the rules $s \longrightarrow t$ such that $s \succ_{\mathcal{T}_S} t$. Note that the set X of previously bound variables is empty. We indeed have failed proving that the ordering $(\succ^X_{\mathcal{T}_S})^+$ is well-founded for an arbitrary X, and we think that it *is not*. As usual in this context, we use Tait and Girard's computability predicate method, with a definition of computability for candidate terms inspired from [4, 1].

Definition 2. *The family of* candidate interpretations $\{[\![\sigma]\!]\}_{\sigma \in \mathcal{T}_S}$ *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 data type and $s :_C \sigma$ is neutral, then $s \in [\![\sigma]\!]$ iff $t \in [\![\tau]\!]$ for all terms t such that $s \succ_{\mathcal{T}_S} t :_C \tau$;

(ii) If σ is a data type and $s = f(\overline{s}) :_C \sigma$ is prealgebraic with $f : \sigma_1 \dots \sigma_n \Rightarrow \sigma' \in \mathcal{F}$, then $s \in [\![\sigma]\!]$ iff $s_i \in [\![\sigma_i]\!]$ for all $i \in Acc(f)$ and $t \in [\![\tau]\!]$ for all terms t such that $s \succ_{\mathcal{T}_S} t :_C \tau$;

(iii) If σ is the functional type $\rho \to \tau$ then $s \in [\![\sigma]\!]$ iff $@(s,t) \in [\![\tau]\!]$ for all $t \in [\![\rho]\!]$;

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.*

Our definition of candidate interpretations is based on a lexicographic combination of an induction on the well-founded type ordering $>^{\rightarrow}_{\mathcal{T}_S}$ (which includes $>_{\mathcal{T}_S}$), and a fixpoint computation for data types. This is so since

(i) the type of the right-hand side term has necessarily decreased strictly in Case 4b: let $s : \sigma$ and $u\{y : \beta \mapsto z : \beta\} : \tau$ be the terms compared in Case 4b, and assume that $s : \sigma \succ^X_{\mathcal{T}_S} t = \lambda y : \beta : u$ is the originating comparison, hence $\sigma \geq_{\mathcal{T}_S} \beta \to \tau$; by Lemma 2, we get $\sigma >_{\mathcal{T}_S} \tau$, showing our claim;

(ii) the type of the right-hand side term has not increased in Case 4a, thanks to the type check.

4.3 Computability properties

We start with a few elementary properties stated without proofs:

Lemma 4. *Assume $\sigma =_{\mathcal{T}_S} \tau$. Then, $[\![\sigma]\!] = [\![\tau]\!]$.*

Lemma 5. *Let $s = @(u,v) :_C \tau$. Then s is computable if u and v are computable.*

Lemma 6. *Let $s :_C \sigma \in \mathcal{T}_{\mathcal{S}}^{min}$ be a strongly normalizable term. Then s is computable.*

Lemma 7. *Assume that $\overline{s}$ is computable and strongly normalizable and that $f(\overline{s}) \rhd_{acc} v$ for some $f \in \mathcal{F} \cup \{@\}$. Then v is computable.*

We now give the fundamental properties of the interpretations. Note that we use our term categorisation to define the computability predicates, and that this is reflected in the computability properties below.

(i) Every computable term is strongly normalizable for $\succ_{\mathcal{T}_{\mathcal{S}}}$;

(ii) If s is a computable candidate term such that $s \succeq_{\mathcal{T}_{\mathcal{S}}} t$, then t is computable;

(iii) A neutral term s is computable iff t is computable for all terms t such that $s \succ_{\mathcal{T}_{\mathcal{S}}} t$;

(iv) An abstraction $\lambda x : \sigma.u$ is computable iff $u\{x \mapsto w\}$ is computable for all computable terms $w :_C \sigma$;

(v) A prealgebraic term $s = f(\overline{s}) :_C \sigma$ such that $f : \overline{\sigma} \to \tau \in \mathcal{F}$ is computable if $\overline{s} :_C \overline{\sigma}$ is computable.

All proofs are adapted from [4], with some additional difficulties. The first four properties are proved together.

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

Given a type σ, we prove by induction on the definition of $[\![\sigma]\!]$ that

(i) Given $s :_C \sigma \in [\![\sigma]\!]$, then s is strongly normalizable;

(ii) Given $s :_C \sigma \in [\![\sigma]\!]$ such that $s \succeq_{\mathcal{T}_{\mathcal{S}}} t$ for some $t :_C \tau$, then $t \in [\![\tau]\!]$;

(iii) A neutral candidate term $u :_C \sigma$ is computable if $w :_C \theta \in [\![\theta]\!]$ for all w such that $u \succ_{\mathcal{T}_{\mathcal{S}}} w$; in particular, variables are computable;

(iv) An abstraction $\lambda x : \alpha.u :_C \sigma$ is computable if $u\{x \mapsto w\}$ is computable for all $w \in [\![\alpha]\!]$.

We prove each property in turn, distinguishing in each case whether σ is a data or functional type.

(ii) 1. Assume that σ is a data type. The result holds by definition of the candidate interpretations.

2. Let $\sigma = \theta \to \rho$. By arrow preservation and decreasingness properties, there are two cases:

   (a) $\rho \geq_{\mathcal{T}_{\mathcal{S}}} \tau$. Let $y :_C \theta \in \mathcal{X}$. By induction hypothesis (iii), $y \in [\![\theta]\!]$, hence $@(s, y) \in [\![\rho]\!]$ by definition of $[\![\sigma]\!]$. Since $@(s, y) :_C \rho \succ_{\mathcal{T}_{\mathcal{S}}} t :_C \tau$ by case 2a of the definition, t is computable by induction hypothesis (ii).

(b) $\tau = \theta' \to \rho'$, with $\theta =_{\mathcal{T}_S} \theta'$ and $\rho \geq_{\mathcal{T}_S} \rho'$. Since s is computable, given $u \in [\![\theta]\!]$, then $@(s,u) \in [\![\rho]\!]$. By monotonicity, $@(s,u) \succ^X_{\mathcal{T}_S} @(t,u)$. By induction hypothesis (ii) $@(t,u) \in [\![\rho']\!]$. Since $[\![\theta]\!] = [\![\theta']\!]$ by Lemma 4, t is computable by definition of $[\![\tau]\!]$.

(i) 1. Assume first that σ is a data type. Let $s \succ_{\mathcal{T}_S} t$. By definition of $[\![\sigma]\!]$, t is computable, hence is strongly normalizable by induction hypothesis. It follows s is strongly normalizable in this case.

2. Assume now that $\sigma = \theta \to \tau$, and let $s_0 = s :_C \sigma = \sigma_0 \succ_{\mathcal{T}_S} s_1 :_C \sigma_1 \ldots \succ_{\mathcal{T}_S} s_n :_C \sigma_n \succ_{\mathcal{T}_S} \ldots$ be a derivation issuing from s. Therefore $s_i \in [\![\sigma_i]\!]$ by induction on i, using the assumption that s is computable for $i = 0$ and otherwise by the already proved property (ii). Such derivations are of the following two kinds:
   (a) $\sigma >_{\mathcal{T}_S} \sigma_i$ for some i, in which case s_i is strongly normalizable by induction hypothesis (i). The derivation issuing from s is therefore finite.
   (b) $\sigma_i =_{\mathcal{T}_S} \sigma$ for all i, in which case $\sigma_i = \theta_i \to \tau_i$ with $\theta_i =_{\mathcal{T}_S} \theta$. Then, $\{@(s_i, y :_C \theta) :_C \tau_i\}_i$ is a sequence of candidate terms which is strictly decreasing with respect to $\succ_{\mathcal{T}_S}$ by monotonicity. Since $y :_C \theta$ is computable by induction hypothesis (iii), $@(s_i, y)$ is computable by definition of $[\![\tau_i]\!]$. By induction hypothesis, the above sequence is finite, implying that the starting sequence itself is finite.

   Therefore, s is strongly normalizing as well in this case.

(iii) 1. Assume that σ is a data type. The result holds by definition of $[\![\sigma]\!]$.

2. Assume now that $\sigma = \sigma_1 \to \sigma_2$. By definition of $[\![\sigma]\!]$, u is computable if the neutral term $@(u, u_1)$ is computable for all $u_1 \in [\![\sigma_1]\!]$. By induction hypothesis, $@(u, u_1)$ is computable iff all its reducts w are computable.

   Since u_1 is strongly normalizable by induction hypothesis (i), we show by induction on the pair $(u_1, |w|)$ ordered by $(\succ_{\mathcal{T}_S}, >_{\mathbb{N}})$ that all reducts w of $@(u, u_1)$ are computable. Since u is neutral, hence is not an abstraction, there are three possible cases:
   (a) $@(u, u_1) \succ_{\mathcal{T}_S} w$ by Case 2a, therefore $u \trianglerighteq_{acc} v \succeq_{\mathcal{T}_S} w$ or $u_1 \trianglerighteq_{acc} v \succeq_{\mathcal{T}_S} w$ for some v. Since the type of w is smaller or equal to the type of $@(u, u_1)$, it is strictly smaller than the type of u, hence $w \neq u$. Therefore, in case $v = u$, w is a reduct of u, hence is computable by assumption. Otherwise, v is u_1 or a minimal-type subterm of u_1, in which case it is computable by assumption on u_1 and Lemma 6, or a minimal-type subterm

of u in which case $u \succ_{\mathcal{T}_S} v$ by Case 1a or 2a since the neutral term u is not an abstraction, and therefore v is computable by assumption. It follows that w is computable by induction hypothesis (ii).

(b) $@(u, u_1) \succ_{\mathcal{T}_S} w$ by Case 2b, therefore $w = @(v, v_1)$ and also $\{u, u_1\}(\succeq_{\mathcal{T}_S})_{mul}\{w_1, w_2\}$. For type reason, there are again two cases:

- w_1 and w_2 are strictly smaller than u, u_1, in which case w_1 and w_2 are computable by assumption or induction hypothesis (ii), hence w is computable by Lemma 5.
- $u = w_1$ and $u_1 \succ_{\mathcal{T}_S} w_2$, implying that w_2 is computable by assumption and induction hypothesis (ii). We then conclude by induction hypothesis since $(u_1, \_)(\succ_{\mathcal{T}_S}, >_{\mathbb{N}})_{lex}(w_2, \_)$.

(c) $@(u, u_1) \succ_{\mathcal{T}_S} w$ by Case 4b, hence $w = \lambda x : \beta.w', x \notin \mathcal{V}ar(w')$ and $@(u, u_1) \succ w'$. By induction hypothesis (iv) and the fact that $x \notin \mathcal{V}ar(w')$, w is computable if w' is computable. Since the type of $\lambda x : \beta.w'$ is strictly bigger than the type of w', we get $@(u, u_1) \succ_{\mathcal{T}_S} w'$. We conclude by induction hypothesis, since $(u_1, \lambda x.w')(\succ_{\mathcal{T}_S}, >_{\mathbb{N}})_{lex}(u_1, w')$.

(iv) By definition of $[\![\sigma]\!]$, the abstraction $\lambda x : \alpha.u :_C \sigma$ is computable if the term $@(\lambda x.u, w)$ is computable for an arbitrary $w \in [\![\alpha]\!]$.
Since variables are computable by induction hypothesis (iii), $u = u\{x \mapsto x\}$ is computable by assumption. By induction hypothesis (i), u and w are strongly normalizable. We therefore prove that $@(\lambda x.u, w)$ is computable by induction on the pair (u, w) compared in the ordering $(\succ_{\mathcal{T}_S}, \succ_{\mathcal{T}_S})_{lex}$.
Since $@(\lambda x.u, w)$ is neutral, we need to show that all reducts v of $@(\lambda x.u, w)$ are computable. We consider the four possible cases in turn:

1. If $@(\lambda x.u, w) \succ_{\mathcal{T}_S} v$ by Case 2a, there are two cases:
   - if $w \succeq_{\mathcal{T}_S} v$, we conclude by induction hypothesis (ii) that v is computable.
   - if $\lambda x.u \succeq_{\mathcal{T}_S} v$, then $\lambda x.u \succ_{\mathcal{T}_S} v$ since the type of $\lambda x.u$ must be strictly bigger than the type of v. There are two cases depending on the latter comparison.
   If the comparison is by Case 3a, then $u \succeq_{\mathcal{T}_S} v$, and we conclude by induction hypothesis (ii) that v is computable.
   If the comparison is by Case 3b, then $v = \lambda x : \alpha'.u'$ with $\alpha =_{\mathcal{T}_S} \alpha'$. By stability, $u\{x \mapsto w\} \succ_{\mathcal{T}_S} u'\{x \mapsto w\}$, hence $u'\{x \mapsto w\}$

is computable by property (ii) for an arbitrary $w \in [\![\alpha]\!] = [\![\alpha']\!]$ by lemma 4. It follows that v is computable by induction hypothesis, since $(u, \_)(\succ_{\mathcal{T}_S}, \succ_{\mathcal{T}_S})_{lex}(u', \_)$.

2. If $@(\lambda x.u, w) \succ_{\mathcal{T}_S} v$ by case 2b, then $v = @(v_1, v_2)$, and by definition of $\succ$, $\{\lambda x.u, w\}(\succ_{\mathcal{T}_S})_{mul}\{v_1, v_2\}$. There are three cases:
   - $v_1 = \lambda x.u$ and $w \succ_{\mathcal{T}_S} v_2$. Then v_2 is computable by induction hypothesis (ii) and, since $u\{x \mapsto v_2\}$ is computable by the main assumption, $@(v_1, v_2)$ is computable by induction hypothesis, since $(\lambda x.u, w)(\succ_{\mathcal{T}_S}, \succ_{\mathcal{T}_S})_{lex}(\lambda x.u, v_2)$.
   - Terms in $\{v_1, v_2\}$ are reducts of u and w. Therefore, v_1 and v_2 are computable by induction hypothesis (ii) and v is computable by Lemma 5.
   - Otherwise, for typing reason, v_1 is a reduct of $\lambda x.u$ of the form $\lambda x.u'$ with $u \succ_{\mathcal{T}_S} u'$, and v_2 is a reduct of the previous kind. By the main assumption, $u\{x \mapsto v''\}$ is computable for an arbitrary computable v''. Besides, $u\{x \mapsto v''\} \succ_{\mathcal{T}_S} u'\{x \mapsto v''\}$ by stability. Therefore $u'\{x \mapsto v''\}$ is computable for an arbitrary computable v'' by induction hypothesis (ii). Then $@(v_1, v_2)$ is computable by induction hypothesis, since $(u, \_)(\succ_{\mathcal{T}_S}, \succ_{\mathcal{T}_S})_{lex}(u', \_)$.
3. If $@(\lambda x.u, w) \succ_{\mathcal{T}_S} v$ by Case 4b, then $v = \lambda x.v'$, $x \notin \mathcal{V}ar(v')$ and $@(\lambda x.u, w) \succ_{\mathcal{T}_S} v'$. Since $\lambda x.v' \succ_{\mathcal{T}_S} v'$ by Case 3a, v' is computable by induction hypothesis. Since $x \notin \mathcal{V}ar(v')$, it follows that $\lambda x.v'$ is computable.
4. If $@(\lambda x.u, w) \succ_{\mathcal{T}_S} v$ by case 2c, then $u\{x \mapsto w\} \succeq_{horpo} v$. By assumption, $u\{x \mapsto w\}$ is computable, and hence v is computable by property (ii). □

We are left with property (v) whose proof differs from [4].

Proof. Property (v). As we have seen, each data type interpretation $[\![\sigma]\!]$ is the least fixpoint of a monotone function G on the powerset of the set of terms. Hence, for every computable term $t \in [\![\sigma]\!]$, there exists a smallest ordinal $o(t)$ such that $t \in G^{o(t)}(\emptyset)$, where G^a is the a transfinite iteration of G. The relation $\sqsupset$, defined by $t \sqsupset u$ iff $o(t) > o(u)$, is a well-founded ordering which is compatible with $\succ_{\mathcal{T}_S}$: if $t \succ_{\mathcal{T}_S} u$ then $t \sqsupseteq u$. The proof is by induction on the type ordering. Therefore, $\succ_{\mathcal{T}_S} \cup \sqsupset$ is well-founded on computable terms. Note that the result would again hold for terms headed by a function symbol with a functional output.

We use this remark to build our outer induction argument: we prove that $f(\overline{s})$ is computable by induction on the pair $(f, \overline{s})$ ordered lexicographically by $(>_{\mathcal{F}}, (\succ_{\mathcal{T}_S} \cup \sqsupset)_{stat_f})_{lex}$. This is our outer statement (OH).

Since $f(\overline{s})$ is prealgebraic, it is computable if every subterm at an accessible position is computable (which follows by assumption) and reducts t of s are computable.

Since $\succ_{\mathcal{T}_S}$ is defined in terms of $\succ^X$, we actually prove by an inner induction on the recursive definition of $\succ^X$ the more general inner statement (IH) that $t\gamma$ is computable for an arbitrary term t such that $f(\overline{s}) \succ^X t$ and computable substitution γ of domain X such that $X \cap \mathcal{V}ar(s) = \emptyset$. Since the identity substitution is computable by property (iii), our inner induction hypothesis implies our outer induction hypothesis.

1. If $f(\overline{s}) \succ^X u$ by Case 4a, Then $u \in X$ and we conclude by assumption on γ that $u\gamma$ is computable.
2. If $f(\overline{s}) \succ^X u$ by Case 1a, then $s_i \trianglerighteq_{acc} t$ for some i and $@(t, \overline{x}) \succeq_{\mathcal{T}_S} u$ for some $\overline{x} \in X$. By assumption on $\overline{s}$ and Lemma 7, t is computable. Since t is a subterm of s and $X \cap \mathcal{V}ar(s) = \emptyset$, then $t\gamma = t$ is computable. It follows that $@(t, \overline{x}\gamma)$ is computable. Thus, by stability, $u\gamma$ is computable.
3. If $f(\overline{s}) \succ^X u$ by case 1b, then $u = g(\overline{u})$, $f =_{\mathcal{F}} g$, $s \succ^X \overline{u}$ and finally $\overline{s}\ (\succ_{\mathcal{T}_S} \cup \triangleright_{acc} \succeq^X_{\mathcal{T}_S})_{stat}\ \overline{u}$. By the inner induction hypothesis, $\overline{u}\gamma$ is computable. Assume now that $s_i : \sigma_i \triangleright_{acc} v$ and $@(v, \overline{x}) : \sigma'_i =_{\mathcal{T}_S} \sigma_i \succeq_{\mathcal{T}_S} u_j$. Using the fact that $X \cap \mathcal{V}ar(s) = \emptyset$, by stability we get $s_i\gamma = s_i \triangleright_{acc} v\gamma = v$ and $@(v, \overline{x})\gamma = @(v, \overline{x}\gamma) : \sigma'_i =_{\mathcal{T}_S} \sigma_i \succeq_{\mathcal{T}_S} u_j\gamma$. Moreover, by definition of computability, $s_i \sqsupseteq @(v, \overline{x}\gamma)$. Therefore, $u\gamma = f(\overline{u}\gamma)$ is computable by the outer induction hypothesis.
4. If $f(\overline{s}) \succ^X_{\mathcal{T}_S} u$ by case 4b, then $u = \lambda x.v$ with $x \notin \mathcal{V}ar(s)$ and $f(\overline{s}) \succ^{X \cup \{x\}} v$. By the inner induction hypothesis, $v(\gamma \cup \{x \mapsto w\})$ is computable for an arbitrary computable w. Assuming without loss of generality that $x \notin \mathcal{R}an(\gamma)$, then $v(\gamma \cup \{x \mapsto w\}) = (v\gamma)\{x \mapsto w\}$. Therefore, $u = \lambda x.v\gamma$ is computable by computability property (iv).
5. If $f(\overline{s}) \succ^X u$ by Case 1c, then $u = g(\overline{u})$ with $g \in \mathcal{F} \cup \{@\}$ and $s \succ^X \overline{u}$. By the inner induction hypothesis, $\overline{u}\gamma$ is computable. We conclude by Lemma 5 in case $g = @$ and by the outer induction hypothesis if $g \in \mathcal{F}$. □

4.4 Strong normalization proof

We are now ready for the strong normalization proof.

Lemma 8. *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 computability property (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 Lemma 5.
4. $t = f(t_1, \ldots, t_n)$. Then $t_i\gamma$ is computable by induction hypothesis, hence $t\gamma$ is computable by computability property (vii). □

The proof of our main theorem follows as a corollary of Lemma 8 when using the identity substitution, and of computability property (i).

5 Conclusion

An implementation of the new definition with examples is available from the web page of the third author (http://www.lsi.upc.es/ albert/term.html).

There are still a few possible improvements that we have not yet explored, such as ordering the abstractions according to their type, increasing the set of accessible terms for applications that satisfy the strict positivity restriction, and showing that the new definition is strictly more general that the general schema when adopting the same type discipline. A more difficult problem to be investigated then is the generalization of this new definition to the calculus of constructions along the lines of [5].

References

1. F. Blanqui. Termination and confluence of higher-order rewrite systems. In *Proc. of the 11th International Conference on Rewriting Techniques and Applications*, volume 1833 of *LNCS*, 2000.
2. F. Blanqui, J.-P. Jouannaud, and A. Rubio. Higher order termination: from Kruskal to computability. In *Proc. LPAR, Phnom Penh, Cambodgia, LNCS 4246*, 2006.
3. Jean-Pierre Jouannaud and Albert Rubio. The higher-order recursive path ordering. In Giuseppe Longo, editor, *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, Trento, Italy, July 1999.
4. Jean-Pierre Jouannaud and Albert Rubio. Polymorphic higher-order recursive path orderings. *Journal of the ACM*, 54(1):1–48, 2007.
5. Daria Walukiewicz-Chrzaszcz. Termination of rewriting in the Calculus of Constructions. In *Proceedings of the Workshop on Logical Frameworks and Meta-languages, Santa Barbara, California*, 2000. Satellite workshop of LICS'2000.