The Higher-Order Computability Path Ordering: the End of a Quest

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

[1] INRIA, Campus Scientifique, BP 239, 54506 Vandœuvre-lès-Nancy Cedex, France
[2] LIX, Projet INRIA TypiCal, École Polytechnique and CNRS, 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. Type comparisons are made only when needed, therefore eliminating the need for the computability closure. Bound variables are handled explicitely, making it possible to handle recursors for arbitrary strictly positive inductive types. This new definition appears indeed to capture the essence of computability arguments *à la Tait and Girard*, therefore explaining the title.

1 Introduction

This paper adresses the problem of automating termination proofs for typed higher-order calculi.

The first attempt we know of goes back to Breazu-Tannen and Gallier [14] and Okada [26]. Following-up a previous work of Breazu-Tannen [13] who considered the confluence of such calculi, both groups of authors showed independently that proving strong normalization of a polymorphic lambda-calculus with (first-order) constants defined by first-order rewrite rules was reducible to the termination proof of the set of rewrite rules: beta-reduction need not be considered. Both work used Girard's method based on *reducibility candidates* -also called sometimes *computability candidates*. These works gave rise to a whole new area, by extending the type discipline, and by extending the kind of rules that could be taken care of.

The type disipline was extended soon later by Barbanera and Dougerthy independently to cover the whole calculus of constructions [1, 17].

Higher-order rewrite rules satisfying the *general schema*, a generalization of Gödel's primitive recursion rules for higher types, were then introduced by Jouannaud and Okada [19, 20] in the case of a polymorphic type discipline. The latter work was then extended first by Barbanera and Fernandez [2, 3] and finally by Barbanera, Fernandez and Geuvers to cover the whole calculus of constructions [4].

It turned out that recursors for *simple* inductive types could already be handled, but arbitrary strict inductive types could not, prompting for an extension of the general schema, which was reformulated for that purpose by Blanqui, Jouannaud and Okada [9]. The new formulation was much more expressive, and was indeed able to handled many more inductive types than originally, by allowing for more expressive rules *at the object level* of the calculus of constructions. The new version of the schema was also much more flexible, but rules were still restricted to operate on object-level terms. The schema was finally extended by Blanqui in a series of papers in order to cover the entire calculus of inductive constructions including strong elimination rules [5, 7, 6]. This required a further generalization of the general schema, by allowing for recursive rules on types.

The definition of the general schema used a precedence on higher-order constants, as does Dershowitz recursive path ordering for first-order terms [16]. This suggested generalizing this ordering to the higher-order case, a work done by the last two authors in the case of a simple type disciplines under the name of HORPO [21]. Comparing two terms with the Higher Order Recursive Path Ordering starts by comparing their types under a given well-founded quasi-ordering on types before to proceed recursively on the structure of the compared terms. There were two variants of the subterm case: in the first, following the recursive path ordering tradition, a subterm of the lefthand side was compared with the whole righthand side; in the second, a term belonging to the computability closure of the lefthand 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.

HORPO was then extended to cover the case of the calculus of constructions by Walukiewicz [30], and to use semantic interpretations of terms instead of a bare precedence on function symbols by Borralleras and Rubio [12]. If was finally improved as well by the two original authors, who included the mechanism of the computability closure originating from [9], and allowed for a restricted polymorphic discipline [22]. The ordering and the computability closure definitions shared a lot in common, raising some expectations for a simpler and yet more expressive definition, as advocated in [10]. These expectations were partly met in [11], where a new, syntax oriented recursive definition was given for HORPO, instead of a pair of mutually inductive definitions for the computability closure and the ordering itself. In contrast with the previous definitions, bound variables were handled explicitely by the ordering, allowing for arbitrary abstractions in the righthand sides.

A third, different line of work was started by van de Pol and Schwichtenberg, who wanted to (semi)-automate termination proofs of higher-order rewrite rules based on higher-order pattern matching, a problem generally considered as harder as the previous one [27, 29, 28]. Related attempts with more automation appear in [25, 23], but were rather unconclusive for practical applications. The general schema was then adapted by Blanqui to cover the case of higher-order pattern matching [6]. Finally, Jouananud and Rubio showed how to turn any well-founded order on higher-order terms including beta and eta, into an well-founded ordering for proving termination of such higher-order rules, and introduced a very simple modification of HORPO to apply this result [24].

A fourth line of work started much later, by extending Aart and Giesl's dependency pairs method to the higher-order case. ... Blanqui, Sato, Sakai.

Finally, a last line of work addresses the question of proving termination of higher-order programs. This is of course a slightly different question, usually adressed by using abstract interpretations. These interpretations may indeed use the general schema or HORPO as a basic ingredient for comparing inputs of a recursive call to those of the call they originate from. This line of work was started by ... Neil Jones, Podelski, and later continued by Blanqui ...

We believe that our quest shall be shown to be useful for all these lines of work, either as a building block, or as a guiding principle.

In this paper, we first slightly improve the definition of our ordering in the very basic case of a simple type discipline, and rename it as the Higher Order Computability Path Ordering. We then adress the treatment of inductive types which remained ad'hoc sor far, therefore concluding our quest thanks to the use of accessibility, a relationship which was shown to generalize the notion of inductive type by Blanqui [6].

2 Higher-Order Algebras

Polymorphic higher-order algebras are introduced in [22]. 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 $s : *^n \rightarrow *$, the set of *types* is generated by the constructor $\rightarrow$ for *functional types*:

$$\begin{array}{c}\mathcal{T}_\mathcal{S} := s(\mathcal{T}_\mathcal{S}^n) \mid (\mathcal{T}_\mathcal{S} \rightarrow \mathcal{T}_\mathcal{S}) \\ \text{for } s : *^n \rightarrow * \ \in \mathcal{S}\end{array}$$

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 equiped 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 $i \neq j$. $\mathcal{V}ar(\Gamma) = \{x_1, \ldots, x_n\}$ is the set of variables of Γ. 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;

Variables:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_\Sigma x : \sigma}$$

Functions:

$$\frac{f : \sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma \in \mathcal{F} \quad \Gamma \vdash_\Sigma t_1 : \sigma_1 \ \ldots \ \Gamma \vdash_\Sigma t_n : \sigma_n}{\Gamma \vdash_\Sigma f(t_1, \ldots, t_n) : \sigma}$$

Abstraction:

$$\frac{\Gamma \cdot \{x : \sigma\} \vdash_\Sigma t : \tau}{\Gamma \vdash_\Sigma (\lambda x : \sigma.t) : \sigma \rightarrow \tau}$$

Application:

$$\frac{\Gamma \vdash_\Sigma s : \sigma \rightarrow \tau \quad \Gamma \vdash_\Sigma t : \sigma}{\Gamma \vdash_\Sigma @(s, t) : \tau}$$

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

2. *Prealgebraic* terms headed by a function symbol, assuming (for the moment) 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 extend to terms by morphism, variable capture being avoided by renaming bound variables when necessary. We use postfixed notation for substitution application.

A rewrite rule is a triple $\Gamma \vdash_\Sigma l \rightarrow r$ 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,

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

The notation $l \rightarrow r \in R$ 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 [22], we show that the rewrite relation generated by $R \cup \{$beta,eta$\}$ can be proved by simply checking that $l > r$ for all $l \rightarrow r \in R$ with some higher-order reduction ordering.

3 The Higher-Order Computability Path Ordering

HOCPO is generated from three basic ingredients: a *type ordering*; 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 higher-order computability path ordering. We define the ordering in two steps, accessibility being used in the second step only. The first ordering is therefore simpler, but the second is much more expressive.

3.1 Basic ingredients

- A quasi-ordering on types $\geq_{\mathcal{T_S}}$ called *the type ordering* satisfying the following properties:
  1. *Well-foundedness*: $>_{\mathcal{T_S}}$ is well-founded;
  2. *Arrow preservation*: $\tau \to \sigma =_{\mathcal{T_S}} \alpha$ iff $\alpha = \tau' \to \sigma'$, $\tau' =_{\mathcal{T_S}} \tau$ and $\sigma =_{\mathcal{T_S}} \sigma'$;
  3. *Arrow decreasingness*: $\tau \to \sigma >_{\mathcal{T_S}} \alpha$ implies $\sigma \geq_{\mathcal{T_S}} \alpha$ or $\alpha = \tau' \to \sigma', \tau' =_{\mathcal{T_S}} \tau$ and $\sigma >_{\mathcal{T_S}} \sigma'$;
  4. *Arrow monotonicity*: $\tau \geq_{\mathcal{T_S}} \sigma$ implies $\alpha \to \tau \geq_{\mathcal{T_S}} \alpha \to \sigma$ and $\tau \to \alpha \geq_{\mathcal{T_S}} \sigma \to \alpha$;
  5. *Stability*: (i) $\sigma >_{\mathcal{T_S}} \tau$ implies $\sigma\xi >_{\mathcal{T_S}} \tau\xi$ for all type substitution ξ;
     (ii) $\sigma =_{\mathcal{T_S}} \tau$ implies $\sigma\xi =_{\mathcal{T_S}} \tau\xi$ for all type substitution ξ.
- 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 [22]:

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

Lemma 2. *Assume $\sigma_1 \to \ldots \to \sigma_m \to \sigma \geq_{\mathcal{T_S}} \tau_1 \to \ldots \to \tau_n \to \tau$, where σ, τ are data types. Then, $\sigma \geq_{\mathcal{T_S}} \tau$ and $\sigma_{i_1} =_{\mathcal{T_S}} \tau_1, \ldots, \sigma_{i_n} =_{\mathcal{T_S}} \tau_n$ for some subsequence $i_1, \ldots, i_n$ of $[1..m]$ which we choose minimal for the lexicographic comparison of strings of integers. We shall write $\overline{\tau} \subseteq \overline{\sigma}$.*

Proof. Straightforward induction on m. □

Lemma 3. *Let $\geq_{\mathcal{T_S}}$ be a quasi-ordering on types such that $>_{\mathcal{T_S}}$ is well-founded, arrow monotonic and arrow preserving. Then, the relation $\geq_{\mathcal{T_S}}^{\to} = (\geq_{\mathcal{T_S}} \cup \rhd_{\to})^*$ is a well-founded quasi-ordering on types extending $\geq_{\mathcal{T_S}}$ and $\rhd_{\to}$, whose equivalence coincides with $=_{\mathcal{T_S}}$.*

3.2 Notations

Our ordering notations are as follows:

– $s \succ^X t$ for the main ordering, with a finite set of variables $X \subset \mathcal{X}$ and the convention that X is omitted when empty;
– $s : \sigma \succ^X_{\mathcal{T}_S} t : \tau$ for $s \succ^X t$ and $\sigma \geq_{\mathcal{T}_S} \tau$;
– $l : \sigma \succ_{\mathcal{T}_S} r : \tau$ as initial call for each $l \to r \in R$;
– $s \succ \overline{t}$ is a shorthand for $s \succ u$ for all $u \in \overline{t}$.

We can now introduce the definition of HOCPO.

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

This new definition schema, which appeared first in [11], incorporates two major innovations with respect to the version of HORPO defined in [22]. The first is that terms can be ordered without requiring that their types are ordered accordingly. This will be the case whenever we can conclude that some recursive call is terminating by using computability arguments rather than an induction on types. Doing so, the ordering inherits directly much of the expressivity of the computability closure. The second is the annotation of the ordering by the set of variables X that were originally bound in the righthand side term, but have become free when taking some subterm. This allows rules 1d and 2b to pull out abstractions from the righthand side regardless of the lefthand side term, meaning that abstractions are smallest in the precedence. An innovation with respect to [11] is rule 3b, which compares abstractions according to their respective types. The precedence on function symbols becomes now total when $>_{\mathcal{F}}$ is total on $\mathcal{F}$.

One may wonder why Case 1c uses recursively the weaker comparison $\overline{s}(\succeq_{\mathcal{T}_S})_{stat_f}\overline{t}$ rather than the stronger one $\overline{s}(\succeq^X_{\mathcal{T}_S})_{stat_f}\overline{t}$. The reason is that the latter would not result into a well-founded ordering as shown now:

Example 1. Non termination.

Let a be a type, and $\{f : a \times a \Rightarrow a, g : (a \rightarrow a) \Rightarrow a\}$ be the signature. Let us consider the following non-terminating rule (its righthand side beta-reduces to its lefthand side in one beta-step):

$$f(g(\lambda x.f(x,x)), g(\lambda x.f(x,x))) \rightarrow @(\lambda x.f(x,x), g(\lambda x.f(x,x)))$$

Let us assume that $f >_{\mathcal{F}} g$ and that f has a multiset status. We now show that the ordering modified as suggested above succeeds with the goal

1. $f(g(\lambda x.f(x,x)), g(\lambda x.f(x,x))) \succ_{\mathcal{T}_S} @(\lambda x.f(x,x), g(\lambda x.f(x,x)))$.

Since type checks are trivial, we will omit them, although the reader will note that there are very few of them indeed. Our goal yields two subgoals by Case 1b:

2. $f(g(\lambda x.f(x,x)), g(\lambda x.f(x,x))) \succ \lambda x.f(x,x)$ and
3. $f(g(\lambda x.f(x,x)), g(\lambda x.f(x,x))) \succ g(\lambda x.f(x,x))$.

Subgoal 2 yields by Case 1d

4. $f(g(\lambda x.f(x,x)), g(\lambda x.f(x,x))) \succ^{\{z\}} f(z,z)$ which yields by Case 1c
5. $f(g(\lambda x.f(x,x)), g(\lambda x.f(x,x))) \succ^{\{z\}} z$ twice, solved by Case 1a and
6. $\{g(\lambda x.f(x,x)), g(\lambda x.f(x,x))\}(\succeq^{\{z\}}_{\mathcal{T}_S})\{z,z\}$ solved by Case 1a applied twice.

We are left with subgoal 3 which yields by Case 1b

7. $f(g(\lambda x.f(x,x)), g(\lambda x.f(x,x))) \succ \lambda x.f(x,x)$, which happens to be the already solved subgoal 2, and we are done.

With the definition we gave, subgoal 6 becomes:
$\{g(\lambda x.f(x,x)), g(\lambda x.f(x,x))\}(\succeq_{\mathcal{T}_S} \cup \succeq_{acc}^{X})_{mul}\{z,z\}$ and does not succeed with $\succeq_{\mathcal{T}_S}$ since the set of previously bound variables has been made empty

The reader can check that chosing the precedence $g >_{\mathcal{F}} f$ yields exactly the same result in both cases. □

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 $\mathbb{N} \rightarrow Ord$, in which Ord occurs positively. As a consequence, the recursor admits a more complex structure than that of natural numbers, with an explicit abstraction in the righthand side of the rule for lim. The strong normalization proof of such recursors is known to be quite hard.

Example 2. Brouwer's ordinals.
$0 : Ord$ $\quad S : Ord \Rightarrow Ord$ $\quad lim : (\mathbb{N} \rightarrow Ord) \Rightarrow Ord$
$n : \mathbb{N}$ $\quad F : \mathbb{N} \rightarrow Ord$
$rec : Ord \times \alpha \times (Ord \rightarrow \alpha \rightarrow \alpha) \times ((\mathbb{N} \rightarrow Ord) \rightarrow (\mathbb{N} \rightarrow \alpha) \rightarrow \alpha) \Rightarrow \alpha$

1. $rec(lim(F), U, X, W) \succ_{\mathcal{T}_S} @(W, F, \lambda n.rec(@(F,n), U, X, W))$ yields 4 subgoals according to Case 1b:
2. $\alpha \geq_{\mathcal{T}_S} \alpha$ 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 1e,
5. $rec(lim(F), U, X, W) \succ F$, which generates by Case 1e the comparison $lim(F) \succ_{\mathcal{T}_S} F$ which fails since $lim(F)$ has a type which is strictly smaller than the type of F.
6. $rec(lim(F), U, X, W) \succ \lambda n.rec(@(F,n), U, X, W)$ which yields by Case 1d
7. $rec(lim(F), U, X, W) \succ^{\{n\}} rec(@(F,n), U, X, W)$ which yields by Case 1c
8. $\{lim(F), U, X, W\}(\succ_{\mathcal{T}_S})_{mul}\{@(F,n), U, X, W\}$, which reduces to
9. $lim(F) \succ_{\mathcal{T}_S} @(F,n)$, whose type comparison succeeds, yielding by Case 1b
10. $lim(F) \succ F$ which succeeds by Case 1e, and
11. $lim(F) \succ n$ which fails because track of n has been lost!

Solving this example requires therefore: first, to access directly the subterm F of $rec(lim(F), U, X, W)$ in order to avoid the type comparison for $lim(F)$ and F when checking whether $rec(lim(F), U, X, W) \succ \{W, F, \lambda n.rec(@(F, n), U$
and second, to keep track of n when comparing $lim(F)$ and n.

3.4 Properties of HOCPO

3.5 Accessibility

While keeping the same type structure, we make use here of a fourth ingredient, the *accessibility* relationship for date types introduced in [5]. This will allows us to solve Brouwer's example, as well as other examples of non-simple inductive types.

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 $=_{\mathcal{T_S}}$), or if $\tau = \rho \rightarrow \theta$ 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 $f : \sigma_1 \ldots \sigma_n \rightarrow \sigma$ 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 $\geq_{\mathcal{T_S}}$, and in case of equivalence (with $=_{\mathcal{T_S}}$), they must occur positively in σ_i. Note that the application operator $@ : (\alpha \rightarrow \beta) \times \alpha \rightarrow \beta$ 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 $\sigma \rightarrow \tau$.

A term u is *accessible* in $f(\overline{s})$, $f \in \mathcal{F}$, iff there exists $i \in Acc(f)$ such that $u = s_i$ or u is *accessible* in s_i. It is accessible in a set $\overline{v}$ iff it is accessible in some $v \in \overline{v}$, in which case we write $\overline{s} \rhd_{acc} u$.

We can now obtain a more elaborated ordering as follows:

Definition 2. $s : \sigma \succ^X t : \tau$ *iff either:*

1. $s = f(\overline{s})$ *with* $f \in \mathcal{F}$ *and either of*
   (a) $t \in X$
   (b) $t = g(\overline{t})$ *with* $f >_{\mathcal{F}} g \in \mathcal{F} \cup \{@\}$ *and* $s \succ^X \overline{t}$
   (c) $t = g(\overline{t})$ *with* $f =_{\mathcal{F}} g \in \mathcal{F}$, $s \succ^X \overline{t}$ *and* $\overline{s}(\succ_{\mathcal{T_S}} \cup \succ^X_{acc})_{stat_f} \overline{t}$
   (d) $t = \lambda y : \beta.w$ *and* $s \succ^{X \cup \{z\}} w\{y \mapsto z\}$ *for* $z : \beta$ *fresh*
   (e) $u \succeq^X_{\mathcal{T_S}} t$ *for some* $u \in \overline{s}$
   (f) $u \succeq^X_{\mathcal{T_S}} t$ *for some* u *such that* $\overline{s} \rhd_{acc} u$
2. $s = @(u, v)$ *and either of*
   (a) $t \in X$

   (b) $t = \lambda y : \beta.w$ *and* $s \succ^{X \cup \{z\}} w\{y \mapsto z\}$
   (c) $t = @(u', v')$ *and* $\{u, v\}(\succ_{\mathcal{T}_\mathcal{S}})_{mul}\{u', v'\}$
   (d) $u \succeq^X_{\mathcal{T}_\mathcal{S}} t$ *or* $v \succeq^X_{\mathcal{T}_\mathcal{S}} t$
   (e) $u = \lambda x : \alpha.w$ *and* $w\{x \mapsto v\} \succeq^X t$

3. $s = \lambda x : \alpha.u$ *and either of*
   (a) $t \in X$
   (b) $t = \lambda y : \beta.w$*,* $\alpha >_{\mathcal{T}_\mathcal{S}} \beta$ *and* $s \succ^X w\{y \mapsto z\}$ *for* $z : \beta$ *fresh*
   (c) $t = \lambda y : \beta.w$*,* $\alpha =_{\mathcal{T}_\mathcal{S}} \beta$ *and* $u\{x \mapsto z\} \succ^X w\{y \mapsto z\}$ *for* $z : \beta$ *fresh*
   (d) $u\{x \mapsto z\} \succeq^X_{\mathcal{T}_\mathcal{S}} t$ *for* $z : \alpha$ *fresh*
   (e) $u = @(v, x)$*,* $x \notin \mathcal{V}ar(v)$ *and* $v \succeq^X t$

The only differences with the previous definition are in Cases 1c of the main definition which uses an additional ordering $\succ^X_{acc}$ *based on the accessibility relationship* $\trianglerighteq_{acc}$ *to compare subterms headed by equivalent function symbols, and in Case 1f which uses the same relationship* $\trianglerighteq_{acc}$ *to reach deep subterms that could not be reached otherwise. Let now*

$$u :_C \overline{\sigma} \to \sigma \succ^X_{acc} t :_C \overline{\tau} \to \tau \textit{ iff}$$
$$\sigma \geq_{\mathcal{T}_\mathcal{S}} \tau,\ \overline{\tau} \subseteq \overline{\sigma} \textit{ and}$$

1. $u \trianglerighteq_{acc} t,$ *or*
2. $t = @(v, \overline{w}), u \trianglerighteq_{acc} v$ *and* $u \succ^X_{\mathcal{T}_\mathcal{S}} \overline{w}$

We could of course strengthen $\succ^X_{acc}$ by giving additional cases, for handling abstractions and function symbols on the right. We do not feel that this is worth it, and leave it as an interesting, albeit non entirely trivial exercise for the interested reader.

We now revisit Brouwer's example, whose strong normalization proof is checked automatically by this new version of the ordering:

Example 3. Brouwer's ordinals.
$0 : Ord$ $S : Ord \Rightarrow Ord$ $lim : (\mathbb{N} \to Ord) \Rightarrow Ord$
$n : \mathbb{N}$ $F : \mathbb{N} \to Ord$
$rec : Ord \times \alpha \times (Ord \to \alpha \to \alpha) \times ((\mathbb{N} \to Ord) \to (\mathbb{N} \to \alpha) \to \alpha) \Rightarrow \alpha$

We skip goals 2,3,4 which do not differ from the previous attempt.

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 1b:
5. $rec(lim(F), U, X, W) \succ F$, which succeeds now by Case 1f,

6. $rec(lim(F), U, X, W) \succ \lambda n.rec(@(F, n), U, X, W)$ which yields by Case 1d
7. $rec(lim(F), U, X, W) \succ^{\{n\}} rec(@(F, n), U, X, W)$ which yields goals 8 and 12 by Case 1c
8. $\{lim(F), U, X, W\}(\succ_{\mathcal{T}_S} \cup \succ^{\{n\}}_{acc})_{mul}\{@(F, n), U, X, W\}$, which reduces to
9. $lim(F) \succ^{\{n\}}_{acc} @(F, n)$ which checks first that $Ord =_{\mathcal{T}_S} Ord$, and then yields successively by Case 2 of $\succ^{\{n\}}_{acc}$:
10. $lim(F) \rhd_{acc} F$ which succeeds since F is accessible in $lim(F)$, and
11. $lim(F) \succeq^{\{n\}}_{\mathcal{T}_S} n$ which succeeds by Case 1a of the main definition (using the type comparison $Ord \geq_{\mathcal{T}_S} \mathbb{N}$). Our remaining goal
12. $rec(lim(F), U, X, W) \succ^{\{n\}} \{@(F, n), U, X, W\}$
    decomposes into three goals trivially solved by Case 1e, that is
13. $rec(lim(F), U, X, W) \succ^{\{n\}} \{U, X, W\}$, and one additional goal
14. $rec(lim(F), U, X, W) \succ^{\{n\}} @(F, n)$ which yields two goals by Case 1b
15. $rec(lim(F), U, X, W) \succ^{\{n\}} F$, which succeeds by Case 1f, and
16. $rec(lim(F), U, X, W) \succ^{\{n\}} n$ which succeeds by Case 1a, therefore ending the computation.

4 Ordering properties

Contrasting with our previous proposal made of an ordering part and a computability closure part, our new ordering is a decidable inductive definition: $s \succ^X t$ and $s \succ^X_{acc} t$ are corecursive definitions 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 when ruling out Case 2e of the definition of $\succ^X$, since all operations used are clearly of linear time complexity. The fact that $\succ^X$ and $\succ^X_{acc}$ are 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.

We proceed proving the various needed properties of our orderings. All proofs are routine.

Lemma 4. *Assume that $u \succeq^X v$ is a successful comparison originating from a comparison of the form $s \succ t$. Then $\mathcal{V}ar(u) \cap X = \emptyset$ and $\mathcal{V}ar(v) \subseteq \mathcal{V}ar(u) \cup X$.*

Lemma 5 (Stability). *$\succeq$ is stable.*

Lemma 6 (Renaming). *If $s \succeq^X t$ and $\xi : X \to \mathcal{X}$ is an injection such that $X\xi \cap \mathcal{V}ar(s) = \emptyset$, then $s\xi \succeq^{X\xi} t\xi$.*

A weakening rule could be added to the ordering to ease the comparisons by conctructing the needed set X rather than guessing it. A strengthening rule could also be added to eliminate useless variables from X.

Lemma 7 (Monotonicity). *$\succeq^X_{\mathcal{T}_S}$ is* monotone *for candidate terms and for terms.*

Lemma 8 (Groundness). *Assume that $s : \sigma$ with σ ground and $s \succeq^X_{\mathcal{T}_S} t : \tau$ with $\forall x : \theta \in X$, then θ is ground. Then, for any successful comparison $u : \delta \succeq^Y v : \rho$ originating from $s \succeq^X_{\mathcal{T}_S} t$, then δ and ρ are ground.*

Proof. First, τ is ground by property of $>_{\mathcal{T}_S}$. The proof is then by induction on $|s| + |t|$.

Lemma 9. *Assume that $f(\overline{s}) \succ^X u$. Then $f(\overline{s}) \succ \lambda X.u$.*

This Lemma does clearly not hold for the ordering $\succ^X_{\mathcal{T}_S}$. This will be a major source of difficulties in our strong normalization proof.

Proof. Simple induction on the size of X and use of Case 1d.

5 Strong normalization

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

Since $\succ_{\mathcal{T}_S}$ is not transitive, this result implies that it is possible (and possibly useful) to replace the recursive call $\overline{s}(\succ_{\mathcal{T}_S} \cup \succ^X_{acc})_{stat_f}\overline{t}$ in Case 1c of the definition of $\succ^X_{\mathcal{T}_S}$ by the more expressive one $\overline{s}((\succ_{\mathcal{T}_S} \cup \succ^X_{acc})^+)_{stat_f}\overline{t}$, to the price of losing decidability of the relation $\succ_{\mathcal{T}_S}$. In practice, we can approximate this more expressive formulation without losing decidability, for example by using the recursive call $\overline{s}(\succ_{\mathcal{T}_S} \cup \succ^X_{acc} \succeq_{\mathcal{T}_S})_{stat_f}\overline{t}$.

We are left with strong normalization, and proceed as in [22] and [11]. One proof is however quite different, that of computability property (v).

5.1 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 $s \in [\![\sigma \rightarrow \tau]\!]$ 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 $\tau' =_{\mathcal{T}_\mathcal{S}} \tau$. Modifying the type system to type terms up to type equivalence $=_{\mathcal{T}_\mathcal{S}}$ is routine [22]. We use the notation $t :_C \sigma$ to indicate that the candidate term t has type σ.

5.2 Candidate interpretations

In the coming sections, we consider the well-foundedness of the strict ordering $(\succ_{\mathcal{T}_\mathcal{S}})^+$, that is, equivalently, the strong normalization of the rewrite relation defined by the rules $s \longrightarrow t$ such that $s \succ_{\mathcal{T}_\mathcal{S}} t$. Note that the set X of previously bound variables is empty. We indeed have failed proving that the ordering $(\succ^X_{\mathcal{T}_\mathcal{S}})^+$ is well-founded for an arbitrary X, and we think that it *is not*, since it cannot be used recursively in Case 1c of our definitions, as shown in Section 3. As usual in this context, we use Tait and Girard's computability predicate method, with a definition of computability for candidate terms inspired from [22, 5].

Definition 3. *The family of* candidate interpretations $\{[\![\sigma]\!]\}_{\sigma \in \mathcal{T}_\mathcal{S}}$ *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 $s \succ_{\mathcal{T}_\mathcal{S}} t :_C \tau$;

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

(iii) If σ is the functional type $\rho \rightarrow \tau$ 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 $>^{\rightarrow}_{\mathcal{T}_\mathcal{S}}$ (which includes $>_{\mathcal{T}_\mathcal{S}}$), and a fixpoint computation for data types. This is so since

(i) the type of the righthand side term has necessarily decreased strictly in Case 1d: let $s : \sigma$ and $u\{y : \beta \mapsto z : \beta\} : \tau$ bne the terms compared in Case 1d, and assume that $s : \sigma \succ^X_{\mathcal{T}_\mathcal{S}} t = \lambda y : \beta : u$ is the originating comparison, hence $\sigma \geq_{\mathcal{T}_\mathcal{S}} \beta \rightarrow \tau$; by Lemma 3, we get $\sigma >_{\mathcal{T}_\mathcal{S}} \tau$, showing our claim;

(ii) the type of the righthand side term has not increased in Case 1a, thanks to the type check.

5.3 Computability properties

We start with a few elementary properties stated without proofs:

Lemma 10. *Assume $\sigma =_{\mathcal{T}_S} \tau$. Then, $[\![\sigma]\!] = [\![\tau]\!]$.*

Lemma 11. *Let $s = @(u, v) :_C \tau$. Then s is computable if u and v are computable.*

Lemma 12. *Let $s :_C \sigma \in \mathcal{T}_S^{min}$ be a strongly normalizable term. Then s is computable.*

Lemma 13. *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 $\succ_{\mathcal{T}_S}$;

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

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

(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 [22], 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 $s \succeq_{\mathcal{T}_S} t$ 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 $u \succ_{\mathcal{T}_S} w$; in particular, variables are computable; note also that w cannot be obtained by Case 1a;

(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 $\sigma = \theta \rightarrow \rho$. 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 2d of the definition, t is computable by induction hypothesis (ii).

   (b) $\tau = \theta' \rightarrow \rho'$, with $\theta =_{\mathcal{T}_\mathcal{S}} \theta'$ and $\rho \geq_{\mathcal{T}_\mathcal{S}} \rho'$. Since s is computable, given $u \in [\![\theta]\!]$, then $@(s, u) \in [\![\rho]\!]$. By monotonicity, $@(s, u) \succ^X_{\mathcal{T}_\mathcal{S}} @(t, u)$. By induction hypothesis (ii) $@(t, u) \in [\![\rho']\!]$. Since $[\![\theta]\!] = [\![\theta']\!]$ by Lemma 10, t is computable by definition of $[\![\tau]\!]$.

(i) 1. Assume first that σ is a data type. Let $s \succ_{\mathcal{T}_\mathcal{S}} t$. 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 $\sigma = \theta \rightarrow \tau$, and let $s_0 = s :_C \sigma = \sigma_0 \succ_{\mathcal{T}_\mathcal{S}} s_1 :_C \sigma_1 \ldots \succ_{\mathcal{T}_\mathcal{S}} s_n :_C \sigma_n \succ_{\mathcal{T}_\mathcal{S}} \ldots$ 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) $\sigma >_{\mathcal{T}_\mathcal{S}} \sigma_i$ for some i, in which case s_i is strongly normalizable by induction hypothesis (i). The derivation issuing from s is therefore finite.

   (b) $\sigma_i =_{\mathcal{T}_\mathcal{S}} \sigma$ for all i, in which case $\sigma_i = \theta_i \rightarrow \tau_i$ with $\theta_i =_{\mathcal{T}_\mathcal{S}} \theta$. Then, $\{@(s_i, y :_C \theta) :_C \tau_i\}_i$ is a sequence of candidate terms which is strictly decreasing with respect to $\succ_{\mathcal{T}_\mathcal{S}}$ 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 (i), 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 $\sigma = \sigma_1 \rightarrow \sigma_2$. 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 2d, 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 12, or a minimal-type subterm of u in which case $u \succ_{\mathcal{T}_S} v$ by Case 1e or 2d 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 2c, 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 11.
- $u = w_1$ and $u_1 \succ_{\mathcal{T}_S} w_2$, implying that w_2 is computable by assumption and induction hypothesis (ii). We 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 2b, 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 2d, 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 3d, then $u \succeq_{\mathcal{T}_S} v$, and we conclude by induction hypothesis (ii) that v is computable.
   If the comparison is by Case 3c, 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 10. 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 2c, 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 11.
   - 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 2b, 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 3d, 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 2e, then $u\{x \mapsto w\} \succeq_{horpo} v$. By assumption, $u\{x \mapsto w\}$ is computable, and hence v is computable by property (ii). $\square$

We are left with property (v) whose proof differs substantially from [22] and even from [11] and needs some preparation.

The interpretation order on candidate terms. As already explained, each data type interpretation $[\![\sigma]\!]$ is the fixpoint of a monotone function F on the powerset of the set of terms. Hence, for every computable term $s \in [\![\sigma]\!]$, there exists some smallest ordinal $o(s)$ such that $s \in F^{o(s)}(\emptyset)$, where F^a is the a transfinite iteration of F. The relation $s \sqsupset_\sigma u$ iff $o(s) > o(u)$, is therefore a well-founded ordering of the set of computable candidate terms. This relation can itself be extended to an ordering of the set of computable candidate terms

$$s :_C \overline{\sigma} \rightarrow \sigma \sqsupset t :_C \overline{\tau} \rightarrow \tau$$
$$\text{iff } \sigma \geq_{\mathcal{T_S}} \tau,\ \overline{\tau} \subseteq \overline{\sigma},\ \text{and for all computable } \overline{u} :_C \overline{\sigma} \text{ and } \overline{v} :_C \overline{\tau} \subseteq \overline{u}$$
$$@(s, \overline{u}) \sqsupset_\sigma @(t, \overline{v})$$

Lemma 14. *$\sqsupset$ is a well-founded ordering of the set of computable candidate terms.*

Proof. This follows easily because any computable candidate term $s :_C \overline{\sigma} \rightarrow \sigma$ can be lifted to a computable candidate term $@(s, \overline{x} : \overline{\sigma}) :_C \sigma$, since variables are computable. □

Lemma 15. *Assume that $s :_C \overline{\sigma} \rightarrow \sigma$ and $t :_C \overline{\tau} \rightarrow \tau$ are computable candidate terms such that $s \succ_{\mathcal{T_S}} t$. Then $s \sqsupset t$.*

Proof. By definition of $\succ_{\mathcal{T_S}}$, $\overline{\sigma} \rightarrow \sigma \geq_{\mathcal{T_S}} \overline{\tau} \rightarrow \tau$, and by Lemma 2, $\sigma \geq_{\mathcal{T_S}} \tau$ and $\overline{\tau} \subseteq \overline{\sigma}$. An easy induction on the length of $\overline{\sigma}$ shows that $@(s, \overline{u}) \succ_{\mathcal{T_S}} @(t, \overline{v})$. Since $@(s, \overline{u})$ and $@(t, \overline{v})$ cannot be abstractions, $@(s, \overline{u}) \sqsupset_\sigma @(t, \overline{v})$ by definition of the interpretations. The result follows. □

Lemma 16. *Let $s = f(\overline{s}) \rhd u = h(\overline{u}) :_C \overline{\sigma} \rightarrow \sigma \succ^X_{acc} t :_C \overline{\tau} \rightarrow \tau$ where σ and τ are data types and $\mathcal{V}ar(u) \cap X = \emptyset$. Assume that the candidate term $u'\gamma$ is computable for all terms u' and computable substitutions γ such that $s \succ^X u'$ and $\mathcal{D}om(\gamma) \subseteq X$, a property called (IH), and that $t' = g(\overline{t'})$ is computable for all computable vectors of candidate terms $\overline{t'}$ such that $(f, \overline{s : \theta})(>_\mathcal{F}, \sqsupset_{stat_f})_{lex}(g, \overline{t' : \rho})$, a property called (OH). Then, $t\gamma$ is computable and $u \sqsupset t\gamma$.*

Proof. By definition $u \succ^X_{acc} t$, $\sigma \geq_{\mathcal{T_S}} \tau$, $\overline{\tau} \subseteq \overline{\sigma}$ and

1. Case 1: $u \rhd_{acc} t$. By (IH), u is computable, hence t is computable by Lemma 13. Further, $u \sqsupset_\sigma t$ by definition of the interpretations. Since $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(u)$ and $\mathcal{V}ar(X) \cap \mathcal{V}ar(u) = \emptyset$, we get $t\gamma = t$ and we are done.
2. Case 2: $t = @(v, w)$, $u \rhd_{acc} v$ and $v \succeq^X_{\mathcal{T}_\mathcal{S}} w$. By Lemma 13 and definition of the interpretations as before, $v = v\gamma$ is computable and $u \sqsupset_\sigma v\gamma$. From $s \rhd u \rhd_{acc} v \succ^X_{\mathcal{T}_\mathcal{S}} w$, we get $s \succ^X_{\mathcal{T}_\mathcal{S}} w$ by Case 1f, hence $w\gamma$ is computable by assumption (IH). Now, since $u \sqsupset_\sigma v\gamma$ and $w\gamma$ is computable, then, by definition of interpretations, $t\gamma = @(v\gamma, w\gamma)$ is a computable candidate term and $u \sqsupset t\gamma$. □

We are now ready for the proof of our last computability property.

Proof. Property (v).

Since $\overline{s}$ is a multiset of computable terms by assumption, $\succ_{\mathcal{T}_\mathcal{S}}$ is well-founded on the set of reducts of terms in $\overline{s}$ by Properties (i) and (ii). We use this remark to build our outer induction argument: we prove that $f(\overline{s})$ is computable by induction on the pair $(f, \overline{s : \sigma})$ ordered lexicographically by $(>_\mathcal{F}, \sqsupset_{stat_f})_{lex}$. This is our outer induction hypothesis (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 $\succ_{\mathcal{T}_\mathcal{S}}$ 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 $u\gamma$ is computable for an arbitrary term u such that $f(\overline{s}) \succ^X u$ 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 (IH) implies our outer induction hypothesis.

1. If $f(\overline{s}) \succ^X u$ by Case 1a, 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 1b, then $u = g(\overline{u})$ with $g \in \mathcal{F} \cup \{@\}$ and $s \succ^X \overline{u}$. By the inner induction hypothesis (IH), $\overline{u}\gamma$ is computable. Since $f >_\mathcal{F} g$, we conclude that $u\gamma$ is computable by (OH).
3. If $f(\overline{s}) \succ^X u$ by case 1c, then $u = g(\overline{u})$, $f =_\mathcal{F} g$, $s \succ^X \overline{u}$ and finally $\overline{s}\ (\succ_{\mathcal{T}_\mathcal{S}} \cup \succ^X_{acc})_{stat}\ \overline{u}$. By the inner induction hypothesis, $\overline{u}\gamma$ is computable. By Lemmas 15 and 16, $\overline{s} \sqsupset_{stat_f} \overline{u}\gamma$. Therefore $u\gamma = f(\overline{u}\gamma)$ is computable by the outer induction hypothesis.
4. If $f(\overline{s}) \succ^X u$ by case 1d, then $u = \lambda x.v$ with $x \notin \mathcal{V}ar(s)$ and $f(\overline{s}) \succ^{X \cup \{x\}} v$. By (IH), $v(\gamma \cup \{x \mapsto w\})$ 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, $u\gamma = \lambda x.v\gamma$ is computable by computability property (iv).

5. If $f(\overline{s}) \succ^X u$ by Case 1e, then $t \succeq^X_{\mathcal{T}_S} u$ for some $t \in \overline{s}$. By assumption on $\overline{s}$, t is computable. Since t is a subterm of s, $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$, hence $\mathcal{V}ar(t) \cap X = \emptyset$, implying that $t\gamma = t$ and $t\gamma$ is therefore computable. By property (iv) $\lambda X.t$ is computable. By mononicity, $\lambda X.t \succ_{\mathcal{T}_S} \lambda X.u$, hence $\lambda X.u$ is computable by Property (ii), and $u\gamma$ is computable by Property (iv).
6. If $f(\overline{s}) \succ^X u$ by Case 1f, then $\overline{s} \rhd_{acc} t \succeq_{\mathcal{T}_S} u$. By Lemma 13, t is computable. By definition of $\rhd_{acc}$, $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$, hence the proof can proceed as previously. □

5.4 Strong normalization proof

We are now ready for the strong normalization proof.

Lemma 17. *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 11.
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 17 when using the identity substitution, and of computability property (i).

6 Conclusion

An implementation of HOCPO with examples is available from the web page of the authors.

There are still a few possible improvements that we have not yet explored, such as ordering the abstractions according to their type, ordering $\mathcal{F} \cup \{@\}$ arbitrarily -this would be useful for some examples, e.g., some versions of Jay's pattern calculus [18], 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 challenging problem to be investigated then is the generalization of this new definition to the calculus of constructions along the lines of [30] and the suggestions made in [22], where an rpo-like ordering on types was proposed which allowed to give a simgle definition for terms and types. Starting with definition 1 is of course desirable.

Finally, it appears that the recursive path ordering and the computing closure are kind of dual of each other: the definitions are quite similar, the closure constructing a set of terms while the ordering deconstructs terms to be compared, the basic case being the same: bound variables and various kinds of subterms (direct, accessible and basic type subterms). Besides, the properties to be satisfied by the type ordering, which were infered from the proof of the computability predicates, almost characterize a recursive path ordering on the first-order type structure. A intriguing, challenging question is therefore to understand the precise relationship between computability predicates and recursive path orderings.

Acknowledgements: the second author wishes to acknowledge the crucial participation of Mitsuhiro Okada to the very beginning of this quest, and to thank Makoto Tatsuta for inviting him in december 2007 at the National Institute for Informatics in Tokyo, whose support provided him with the ressources, peace and impetus to conclude this quest with his coauthors.

References

1. F. Barbanera. Adding algebraic rewriting to the calculus of constructions: Strong normalization preserved. In *Proc. of the 2nd Int. Workshop on Conditional and Typed Rewriting*, 1990.
2. F. Barbanera and M. Fernández. Combining first and higher order rewrite systems with type assignment systems. In *Proc. of the 1st Int. Conf. on Typed Lambda Calculi and Applications*, LNCS 664, 1993.
3. F. Barbanera and M. Fernández. Modularity of termination and confluence in combinations of rewrite systems with λ_ω. In *Proc. of the 20th Int. Colloq. on Automata, Languages, and Programming*, LNCS 700, 1993.

4. F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalization and confluence in the algebraic-λ-cube. In *Proc. of the 9th Symp. on Logic in Computer Science*, IEEE Computer Society, 1994.
5. F. Blanqui. Termination and confluence of higher-order rewrite systems. In *Proc. of the 11th Int. Conf. on Rewriting Techniques and Applications*, volume 1833 of *LNCS*, 2000.
6. F. Blanqui. Inductive Types in the Calculus of Constructions. In TLCA, *Lecture Notes in Computer Science* 2701:395–409. Springer-Verlag, 2003.
7. F. Blanqui. Definitions by rewriting in the calculus of constructions, In LICS, 2001.
8. 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.
9. F. Blanqui, J.-P. Jouannaud, and M. Okada. Inductive Data Types. *Theoretical Computer Science* 277:41–68, 2002.
10. F. Blanqui, J.-P. Jouannaud, and A. Rubio. Higher order termination: from Kruskal to computability. In *Proc. LPAR, Phnom Penh, Cambodgia, LNCS 4246*, 2006.
11. F. Blanqui, J.-P. Jouannaud, and A. Rubio. HORPO with Computability Closure: A Reconstruction In *Proc. LPAR, Yerevan, Armenia, LNCS* , 2007.
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. V. Breazu-Tannen. Combining algebra and higher-order types. In *Proc. of the 3rd Symp. on Logic in Computer Science*, IEEE Computer Society, 1988.
14. V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. In *Proc. of the 16th Int. Colloq. on Automata, Languages, and Programming*, LNCS 372, 1989.
15. V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 83(1), 1991.
16. N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, March 1982.
17. Daniel J. Dougherty. Adding algebraic rewriting to the untyped lambda calculus. Research report, Wesleyan University, USA, 1990.
18. Barry Jay. The Pattern Calculus. To be published.
19. 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.
20. J-P. Jouannaud and M. Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349–391, 1997.
21. 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.
22. Jean-Pierre Jouannaud and Albert Rubio. Polymorphic higher-order recursive path orderings. *Journal of the ACM*, 2007.
23. 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.
24. 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.
25. 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.
26. M. Okada. Strong normalizability for the combined system of the typed lambda calculus and an arbitrary convergent term rewrite system. In *Proc. of the 1989 Int. Symp. on Symbolic and Algebraic Computation*, ACM Press.
27. 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.

28. J. van de Pol. *Termination of Higher-Order Rewrite Systems*. PhD thesis, Utrecht University, The Netherlands, 1996.
29. 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.
30. D. Walukiewicz-Chrzaszcz. Termination of rewriting in the Calculus of Constructions. *Journal Functional Programming*, 13(2):339–414, 2003.