The 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. In this paper, we first briefly survey automated termination proof methods for higher-order calculi. We then concentrate on the higher-order recursive path ordering, for which we provide an improved definition, the Computability Path Ordering. This new definition appears indeed to capture the essence of computability arguments *à la Tait and Girard*, therefore explaining the name of the improved ordering.

1 Introduction

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

The first attempt we know of goes back to Breazu-Tannen and Gallier [24] and Okada [44]. Following up a pioneering work of Breazu-Tannen who considered the confluence of such calculi [23], 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 works used Girard's method based on *reducibility candidates* -also called sometimes *computability predicates*. They then 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 discipline was extended soon later independently by Barbanera and Dougerthy in order to cover the whole calculus of constructions [3, 28].

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 [34, 35] in the case of a polymorphic type discipline. The latter work was then extended first by Barbanera and Fernandez [4, 5] and finally by Barbanera, Fernandez and Geuvers to cover the whole calculus of constructions [6].

It turned out that recursors for *simple* inductive types could be taken care of by the general schema, but arbitrary strict inductive types could not, prompting for an extension of the schema, which was reformulated for that purpose by Blanqui, Jouannaud and Okada [16]. This new formulation was based on the notion of *computability closure* of a term $f(\overline{s})$ headed by a higher-order constant f, defined as a set containing the immediate subterms $\overline{s}$ of $f(\overline{s})$ and closed under computability preserving operations in the sense of Tait and Girard. Membership to the general schema was then defined for an arbitrary rewrite rule as membership of its right-hand side to the computability closure of its left-hand side.

Besides being elegant, this formulation was indeed much more flexible and powerful. By allowing for more expressive rules *at the object level* of the calculus of constructions, it could handle many more inductive types than originally. The general schema was finally extended by Blanqui in a series of papers by allowing for *recursive rules on types*, in order to cover the entire calculus of inductive constructions including strong elimination rules [13, 14].

The definition of the general schema used a precedence on higher-order constants, as does Dershowitz recursive path ordering for first-order terms [26]. This suggested generalizing this ordering to the higher-order case, a work done by Jouannaud and Rubio in the case of a simple type discipline under the name of HORPO [37]. Comparing two terms with HORPO 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, comparing first in the precedence the higher-order constants heading both terms. Following the recursive path ordering tradition, a subterm of the left-hand side could also be compared with the whole right-hand side, regardless of the precedence on their heads.

HORPO was then extended to cover the case of the calculus of constructions by Walukiewicz [51], and to use semantic interpretations of terms instead of a precedence on function symbols by Borralleras and Rubio [21]. HORPO was also improved by the two original authors in two different ways: by comparing in the so-called subterm case an arbitrary term belonging to the computability closure of the left-hand side term with the right-hand side term, therefore generalizing both HORPO and the general schema; and by allowing for a restricted polymorphic discipline [40]. An axiomatic presentation of the rules underlying HORPO can be found in [31]. A more recent work in the same direction is [27].

The ordering and the computability closure definitions turn out to share many similar constructs, raising expectations for a simpler and yet more expressive definition, instead of a pair of mutually inductive definitions for the computability closure and the ordering itself, as advocated in [17]. These expectations were partly met, on the one hand in [15] with a single computability oriented definition, and on the other hand in [18] where a new, syntax oriented recursive definition was given for HORPO. In contrast with the previous definitions, bound variables were handled explicitly by the ordering, allowing for arbitrary abstractions in the right-hand sides.

A third, different line of work was started by van de Pol and Schwichtenberg, who aimed at (semi)-automating termination proofs of higher-order rewrite rules based on higher-order pattern matching, a problem generally considered as harder as the previous one [47, 49, 48]. Related attempts with more automation appear in [43, 38], but were rather unconclusive for practical applications. The general schema was then adapted by Blanqui to cover the case of higher-order pattern matching [11]. Finally, Jouannaud and Rubio showed how to turn any well-founded ordering on higher-order terms including beta and eta, into a well-founded ordering for proving termination of such higher-order rules, and introduced a very simple modification of HORPO as an application of this result [36].

A fourth line of work was started by Borralleras and Rubio. Among other material, Borralleras thesis [20] contained a constraint-based approach to the semantic path ordering [41] which was shown to encompass the dependency pairs method of Arts and Giesl [2, 30] in all its various aspects. Besides the thesis itself, the principles underlying this work are also described in [21] and [22]. An interesting aspect is that they lift to the higher-order case. Extending the dependency pairs method to the higher-order case was also considered independently by Sakai *et al* [46, 45] and Blanqui [10].

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 addressed 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 includes [32, 25, 8, 52, 1, 7, 12, 29]. An important related work, considering pure lambda terms, is [19].

We believe that our quest shall be shown 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 HORPO in the very basic case of a simple type discipline, and rename it as the Computability Path Ordering. We then address the treatment of inductive types which remained *ad hoc* so 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 [13, 14]. We finally list which are the most important question to be addressed for those who would like to start a new quest.

2 Higher-Order Algebras

Polymorphic higher-order algebras are introduced in [40]. 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.

We give here the minimal set of notions to be reasonably self-contained.

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*:

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

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 $\mathcal{F} = \biguplus_{\sigma_1,\ldots,\sigma_n,\sigma} \mathcal{F}_{\sigma_1\times\ldots\times\sigma_n\to\sigma}$. The membership of a given function symbol f to $\mathcal{F}_{\sigma_1\times\ldots\times\sigma_n\to\sigma}$ is called a *type declaration* and written $f : \sigma_1 \times \ldots \times \sigma_n \to \sigma$.

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 $t[u]_p$.

Typable raw terms are called *terms*. The typing judgements are standard. We categorize terms into three disjoint classes:

1. *Abstractions* headed by λ;
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.

Substitutions, rewrite rules and higher-order reduction orderings are as expected, see [40].

3 The Computability Path Ordering

CPO is generated from three basic ingredients: a *type ordering*; a *precedence* on functions symbols; and a *status* for the function symbols. Accessibility is an additionnal ingredient originating in inductive types, while the other three were already needed for defining HORPO. We describe these ingredients before defining the computability path ordering. We define the ordering in two steps, accessibility being used in the second step only. The first ordering is therefore simpler, while the second is more expressive.

3.1 Basic ingredients

- 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 $@ \in Mul$.
- and a quasi-ordering on types $\geq_{\mathcal{T}_\mathcal{S}}$ called *the type ordering* satisfying the following properties, where $=_{\mathcal{T}_\mathcal{S}}$ denotes its associated equivalence relation $\geq_{\mathcal{T}_\mathcal{S}} \cap \leq_{\mathcal{T}_\mathcal{S}}$ and $>_{\mathcal{T}_\mathcal{S}}$ its strict part $\geq_{\mathcal{T}_\mathcal{S}} \setminus \leq_{\mathcal{T}_\mathcal{S}}$:
  1. *Well-foundedness*: $>^{\rightarrow}_{\mathcal{T}_\mathcal{S}} = >_{\mathcal{T}_\mathcal{S}} \cup \rhd_{\rightarrow}$ is well-founded, where $\sigma \rightarrow \tau \rhd_{\rightarrow} \sigma$;
  2. *Right arrow subterm*: $\sigma \rightarrow \tau >_{\mathcal{T}_\mathcal{S}} \tau$;

3. *Arrow preservation*: $\tau \to \sigma =_{\mathcal{TS}} \alpha$ iff $\alpha = \tau' \to \sigma'$, $\tau' =_{\mathcal{TS}} \tau$ and $\sigma =_{\mathcal{TS}} \sigma'$;
4. *Arrow decreasingness*: $\tau \to \sigma >_{\mathcal{TS}} \alpha$ implies $\sigma \geq_{\mathcal{TS}} \alpha$ or else $\alpha = \tau' \to \sigma'$, $\tau' =_{\mathcal{TS}} \tau$ and $\sigma >_{\mathcal{TS}} \sigma'$;

Arrow preservation and decreasingness imply that the type ordering does *not*, in general, have the left arrow subterm property: $\sigma \to \tau \not\geq_{\mathcal{TS}} \sigma$. A first axiomatic definition of the type ordering was given in [39], which did not need right arrow subterm. A new one, expected to be easier to understand, was given in [40] based solely on $\geq_{\mathcal{TS}}$, which uses another axiom, *arrow monotonicity*, to force the right arrow subterm property. As pointed out to us recently, this set of axioms is unfortunately inconsistent [50]. However, the restriction of the recursive path ordering proposed there for a type ordering does not satisfy arrow monotonicity, but does satisfy instead the corrected set of axioms given here.

We now give two important properties of the type ordering:

Lemma 1. *[40] Assuming $\sigma =_{\mathcal{TS}} \tau$, σ is a data type iff τ is a data type.*

Lemma 2. *If $\alpha \to \sigma \geq_{\mathcal{TS}} \beta \to \tau$ then $\sigma \geq_{\mathcal{TS}} \tau$.*

Proof. If $\alpha \to \sigma =_{\mathcal{TS}} \beta \to \tau$ then, by arrow preservation, $\alpha =_{\mathcal{TS}} \beta$ and $\sigma =_{\mathcal{TS}} \tau$. If $\alpha \to \sigma >_{\mathcal{TS}} \beta \to \tau$, then, by arrow decreasingness, either $\alpha =_{\mathcal{TS}} \beta$ and $\sigma >_{\mathcal{TS}} \tau$, or else $\sigma >_{\mathcal{TS}} \beta \to \tau$. In the latter case, $\beta \to \tau >_{\mathcal{TS}} \tau$ by right arrow subterm and we conclude by transitivity. □

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;
- $s : \sigma \succ^X_{\mathcal{TS}} t : \tau$ for $s \succ^X t$ and $\sigma \geq_{\mathcal{TS}} \tau$;
- $l : \sigma \succ_{\mathcal{TS}} r : \tau$ as initial call for each $l \to r \in R$;
- $s \succ \bar{t}$ is a shorthand for $s \succ u$ for all $u \in \bar{t}$;
- $\succeq$ is the reflexive closure of $\succ$.

We can now introduce the definition of CPO.

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

Because function symbols, applications and abstractions do not behave exactly the same, we chosed to organize the definition according to the left-hand side head symbol: a function symbol, an application, or an abstraction successively. In all three cases, we first take care of the case where the right-hand side is a bound variable -case named *variable*-, then headed by a symbol which is the same as (or equivalent to) the left-hand side head symbol -case *status*-, or headed by a symbol which is strictly smaller in the precedence than the left-hand side head symbol -case *precedence*-, before to go with the -case *subterm*. The precedence case breaks into two sub-cases when the left-hand side is a function symbol, because abstractions, which can be seen as smaller than other symbols, need renaming of their bound variable when pulled out, which makes their treatment a little bit different formally from the standard precedence case. There are two specific cases for *application* and *abstraction*: one for beta-reduction, and one for eta-reduction, which are both built in the definition.

This new definition schema appeared first in [18] in a slightly different format. It incorporates two major innovations with respect to the

version of HORPO defined in [40]. 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 schema used in [40]. The second 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 when taking some subterm. This allows rules 1d, 2c and 3c to pull out abstractions from the right-hand side regardless of the left-hand side term, meaning that abstractions are smallest in the precedence. Among the innovations with respect to [18] are rules 3c, which compares abstractions whose bound variables have non-equivalent types, and rule 2d, whose formulation is now stronger.

This definition suffers some subtle limitations:

1. Case 1d uses recursively the comparison $s \succ^{X \cup \{z\}} w\{y \mapsto z\}$ for z fresh, implying that the occurrences of z in w can be later taken care of by Case 1a, 2a or 3a. This is no limitation.
   Cases 2c and 3c use instead the recursive comparison $s \succ^{X} w\{y \mapsto z\}$, with z fresh, hence $z \notin X$. As a consequence, these recursive calls cannot succeed if $z \in \mathcal{V}ar(w)$. We could have added this redundant condition for sake of clarity. We prefered to privilege uniformity and locality of tests.
   As a consequence, Cases 1d, 2c and 3c cannot be packed together as it was unfortunately done in [18], where correct proofs were however given which did of course not correspond to the given definition.
2. The subterm case 1e uses recursively the comparison $u \succ_{\mathcal{T}_\mathcal{S}} t$ instead of the expected comparison $u \succ^{X}_{\mathcal{T}_\mathcal{S}} t$.
   On the other hand, the other subterm definitions, Cases 2d and 3d use the expected comparisons $u \succeq^{X}_{\mathcal{T}_\mathcal{S}} t$ or $v \succeq^{X}_{\mathcal{T}_\mathcal{S}} t$ in the first case, and $u\{x \mapsto z\} \succeq^{X}_{\mathcal{T}_\mathcal{S}} t$ in the second. This implies again that the various subterm cases cannot be packed together.
3. Case 1b uses recursively the comparison $\overline{s}(\succ_{\mathcal{T}_\mathcal{S}})_{stat_f}\overline{t}$ instead of the stronger comparison $\overline{s}(\succ^{X}_{\mathcal{T}_\mathcal{S}})_{stat_f}\overline{t}$.

All our restrictions are justified by their use in the well-foundedness proof of $\succ_{\mathcal{T}_\mathcal{S}}$. There is an even better argument: the ordering would not be well-founded otherwise, as can be shown by means of counter-examples. We give two below.

We start with an example of non-termination obtained when replacing the recursive call $\overline{s}(\succ_{\mathcal{T}_\mathcal{S}})_{stat_f}\overline{t}$ by $\overline{s}(\succ^X_{\mathcal{T}_\mathcal{S}})_{stat_f}\overline{t}$ in Case 1b.

Example 1. Let a be a type, and $\{f : a \times a \to a, g : (a \to a) \to a\}$ be the signature. Let us consider the following non-terminating rule (its right-hand side beta-reduces to its left-hand side in one beta-step):

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

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

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 sub-goals by Case 1c:

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

Sub-goal 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 1b
5. $f(g(\lambda x.f(x,x)), g(\lambda x.f(x,x))) \succ^{\{z\}} z$ twice, solved by Case 1a and
6. $\{g(\lambda x.f(x,x)), g(\lambda x.f(x,x))\}(\succeq^{\{z\}}_{\mathcal{T}_\mathcal{S}})\{z,z\}$ solved by Case 1a applied twice.

We are left with sub-goal 3 which yields by Case 1c

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 sub-goal 2, and we are done.

With the definition we gave, sub-goal 6 becomes:
$\{g(\lambda x.f(x,x)), g(\lambda x.f(x,x))\}(\succ_{\mathcal{T}_\mathcal{S}})_{mul}\{z,z\}$ and does not succeed 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. □

Next is an example of non-termination due to Cynthia Kop and Femke van Raamsdong [50], obtained when replacing the recursive call $s \succ^X w\{y \mapsto z\}$ by $s \succ^{X\cup\{z\}} w\{y \mapsto z\}$ in Case 2c.

Example 2. Let o be a type, and $\{f : o \to o, A : o, B : o \to o \to o\}$ be the signature. Let us consider the following non-terminating set of rules:

$$@(@(B, A), A) \to @(\lambda z : o.f(z), A) \qquad (1)$$
$$f(A) \to @(@(B, A), A) \qquad (2)$$

since

$$@(@(B, A), A) \underset{1}{\longrightarrow} @(\lambda z : o.f(z), A) \underset{\beta}{\longrightarrow} f(A) \underset{2}{\longrightarrow} @(@(B, A), A)$$

Let us assume that $A >_{\mathcal{F}} f >_{\mathcal{F}} B$ and consider the goals:

1. $@(@(B, A), A) : o \succ_{\mathcal{T}_{\mathcal{S}}} @(\lambda z : o.f(z), A) : o$, and
2. $f(A) : o \succ_{\mathcal{T}_{\mathcal{S}}} @(@(B, A), A) : o$.

Goal 1 yields two sub-goals by Case 2b:

3. $A : o \succeq_{\mathcal{T}_{\mathcal{S}}} A : o$, which succeeds trivially, and
4. $@(B, A) : o \to o \succ_{\mathcal{T}_{\mathcal{S}}} \lambda z : o.f(z) : o \to o$ which yields by modified Case 2c:
5. $@(B, A) \succ^{\{z\}} f(z)$, which yields in turn by Case 2d
6. $A : o \succ^{\{z\}}_{\mathcal{T}_{\mathcal{S}}} f(z) : o$ which yields by Case 1c
7. $A : o \succ^{\{z\}}_{\mathcal{T}_{\mathcal{S}}} z : o$ which succeeds by Case 1a.

Note that we have used B for its large type, and A for eliminating $f(z)$, exploiting a kind of divide and conquer ability of the ordering. We are left with goal 2 which yields two subgoals by Case 1d

8. $f(A) \succ A$ which succeeds by Case 1e, and
9. $f(A) \succ @(B, A)$, which yields by Case 1c:
10. $f(A) \succ A$, which succeeds by Case 1e, and
11. $f(A) \succ B$, which succeeds by Case 1c, therefore ending the computation. □

More examples justifying our claim that the quest has come to en end are given in the full version of this paper.

We give now an example of use of the computability path ordering 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 $\mathbb{N} \to O$, in which O occurs positively. As a consequence, the recursor admits a more complex structure than that of natural numbers, with an explicit abstraction in the right-hand side of the rule for lim. The strong normalization proof of such recursors is known to be hard.

Example 3. Brouwer's ordinals.
$0 : O \qquad S : O \to O \qquad lim : (\mathbb{N} \to O) \to O$
$rec : O \times \alpha \times (O \to \alpha \to \alpha) \times ((\mathbb{N} \to O) \to (\mathbb{N} \to \alpha) \to \alpha) \to \alpha$

The rules defining the recursor on Brouwer's ordinals are:

$rec(0, U, X, W) \to U$
$rec(S(n), U, X, W) \to @(X, n, rec(n, U, X, W))$
$rec(lim(F), U, X, W) \to @(W, F, \lambda n.rec(@(F, n), U, X, W))$

Let us try to prove that the third rule is in $\succ_{\mathcal{T}_\mathcal{S}}$.

1. $s = rec(lim(F), U, X, W) \succ_{\mathcal{T}_\mathcal{S}} @(W, F, \lambda n.rec(@(F, n), U, X, W))$ yields 4 sub-goals according to Case 1c:
2. $\alpha \geq_{\mathcal{T}_\mathcal{S}} \alpha$ which is trivially satisfied, and
3. $s \succ \{W, F, \lambda n.rec(@(F, n), U, X, W)\}$ which simplifies to:
4. $s \succ W$ which succeeds by Case 1e,
5. $s \succ F$, which generates by Case 1e the comparison $lim(F) \succ_{\mathcal{T}_\mathcal{S}} F$ which fails since $lim(F)$ has a type which is strictly smaller than the type of F.
6. $s \succ \lambda n.rec(@(F, n), U, X, W)$ which yields by Case 1d
7. $s \succ^{\{n\}} rec(@(F, n), U, X, W)$ which yields by Case 1b
8. $\{lim(F), U, X, W\}(\succ_{\mathcal{T}_\mathcal{S}})_{mul}\{@(F, n), U, X, W\}$, which reduces to
9. $lim(F) \succ_{\mathcal{T}_\mathcal{S}} @(F, n)$, whose type comparison succeeds, yielding by Case 1c
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 s in order to avoid the type comparison for $lim(F)$ and F when checking recursively whether the comparison $s \succ \lambda n.rec(@(F, n), U, X, W)$ holds; and second, to keep track of n when comparing $lim(F)$ and n.

3.4 Accessibility

While keeping the same type structure, we make use here of a fourth ingredient, the *accessibility* relationship for data types introduced in [11]. This will allow us to solve Brouwer's example, as well as other examples of non-simple inductive types.

We say that a data type is *simple* is it is a type constant. We restrict here our definition of accessibility to simple data types. To this end, we assume that all type constructors are constants, that is, have arity zero.

We can actually do a little bit more, assuming that simple data types are not greater or equal (in $\geq_{\mathcal{TS}}$) to non-constant data types, allowing the simple data types to live in a separate world.

The sets of *positive and negative positions* in a type σ are inductively defined as follows:

- $Pos^+(\sigma) = \{\varepsilon\}$ if σ is a simple data type
- $Pos^-(\sigma) = \emptyset$ if σ is a simple data type
- $Pos^\delta(\sigma \to \tau) = 1 \cdot Pos^{-\delta}(\sigma) \cup 2 \cdot Pos^\delta(\tau)$
  where $\delta \in \{+, -\}$, $-+ = -$ and $-- = +$ (usual rules of signs)

Then we say that a simple data type σ occurs (only) *positively* in a type τ if it occurs only at positive positions: $Pos(\sigma, \tau) \subseteq Pos^+(\tau)$, where $Pos(\sigma, \tau)$ is the set of positions of the occurrences of σ in τ.

The set $Acc(f)$ of *accessible argument positions* of a function symbol $f : \sigma_1 \dots \sigma_n \to \sigma$, where σ is a simple data type, is the set of integers $i \in \{1, \dots, n\}$ such that:

- no simple data type greater than σ occurs in σ_i,
- simple data types equivalent to σ occurs only positively in σ_i.

Then a term u is *accessible* in a term v, written $v \rhd_{acc} u$, iff v is a pre-algebraic term $f(\overline{s})$ and there exists $i \in Acc(f)$ such that either $u = s_i$ or u is accessible in s_i ($\rhd_{acc}$ is transitive).

A term u is accessible in a sequence of terms $\overline{v}$ iff it is accessible in some $v \in \overline{v}$, in which case we write $\overline{s} \rhd_{acc} u$. Note that the terms accessible in a term v are strict subterms of v.

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* $f =_{\mathcal{F}} g \in \mathcal{F}$, $s \succ^X \overline{t}$ *and* $\overline{s} (\succ_{\mathcal{TS}} \cup \succ_{acc}^{X,s})_{stat_f} \overline{t}$
   (c) $t = g(\overline{t})$ *with* $f >_{\mathcal{F}} g \in \mathcal{F} \cup \{@\}$ *and* $s \succ^X \overline{t}$
   (d) $t = \lambda y : \beta . w$ *and* $s \succ^{X \cup \{z\}} w\{y \mapsto z\}$ *for* $z : \beta$ *fresh*
   (e) $u \succeq_{\mathcal{TS}} t$ *for some* $u \in \overline{s}$
   (f) $u \succeq_{\mathcal{TS}} t$ *for some* u *such that* $\overline{s} \rhd_{acc} u$
2. *$s = @(u, v)$ and either of*
   (a) $t \in X$
   (b) $t = @(u', v')$ *and* $\{u, v\} (\succ_{\mathcal{TS}})_{mul} \{u', v'\}$

(c) $t = \lambda y : \beta.w$ *and* $s \succ^X w\{y \mapsto z\}$ *for* $z : \beta$ *fresh*
(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* $u\{x \mapsto z\} \succ^X w\{y \mapsto z\}$ *for* $z : \beta$ *fresh*
(c) $t = \lambda y : \beta.w$, $\alpha \neq_{\mathcal{T}_S} \beta$ *and* $s \succ^X w\{y \mapsto z\}$ *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* $v \succeq^X t$

where $u : \sigma \succ^{X,s}_{acc} t : \tau$ *iff* $\sigma \geq_{\mathcal{T}_S} \tau$, $t = @(v, \overline{w})$, $u \rhd_{acc} v$ *and* $s \succ^X \overline{w}$.

The only differences with the previous definition are in Case 1b of the main definition which uses an additional ordering $\succ^{X,s}_{acc}$ based on the accessibility relationship $\rhd_{acc}$ to compare subterms headed by equivalent function symbols, and in Case 1f which uses the same relationship $\rhd_{acc}$ to reach deep subterms that could not be reached otherwise. Following up a previous discussion, notice that we have kept the same formulation in Cases 2c and 3c, rather than use the easier condition $y \notin \mathcal{V}ar(w)$.

We could of course strengthen $\succ^{X,s}_{acc}$ by giving additional cases, for handling abstractions and function symbols on the right [11, 15]. We could also think of improving Case 1e by replacing $\overline{s} \rhd_{acc} u$ by the stronger condition $\overline{s} \succ^{X,s}_{acc} u$. We have not tried these improvements yet.

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

Example 4. Brouwer's ordinals.

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

1. $s = rec(lim(F), U, X, W) \succ_{\mathcal{T}_S} @(W, F, \lambda n.rec(@(F, n), U, X, W))$ yields 4 sub-goals according to Case 1c:
5. $s \succ F$, which succeeds now by Case 1f,
6. $s \succ \lambda n.rec(@(F, n), U, X, W)$ which yields by Case 1d
7. $s \succ^{\{n\}} rec(@(F, n), U, X, W)$ which yields goals 8 and 12 by Case 1b
8. $\{lim(F), U, X, W\}(\succ_{\mathcal{T}_S} \cup \succ^{\{n\},s}_{acc})_{mul}\{@(F, n), U, X, W\}$, which reduces to
9. $lim(F) \succ^{\{n\},s}_{acc} @(F, n)$ which succeeds since $O =_{\mathcal{T}_S} O$, F is accessible in $lim(F)$ and $s \succ^{\{n\}} n$ by case Case 1a. Our remaining goal
10. $s \succ^{\{n\}} \{@(F, n), U, X, W\}$
decomposes into three goals trivially solved by Case 1e, that is
11. $s \succ^{\{n\}} \{U, X, W\}$, and one additional goal

12. $s \succ^{\{n\}} @(F, n)$ which yields two goals by Case 1c
13. $s \succ^{\{n\}} F$, which succeeds by Case 1f, and
14. $s \succ^{\{n\}} n$ which succeeds by Case 1a, thus ending the computation.

4 Conclusion

An implementation of CPO with examples is available from the web page of the third author.

There is still a couple of possible improvements that deserve to be explored thoroughly: change -if possible at all- the recursive calls of Cases 1e, 2c and 3c of the definition of CPO as discussed in Section 3; ordering $\mathcal{F} \cup \{@\}$ arbitrarily -this would be useful for some examples, e.g., some versions of Jay's pattern calculus [33]; increasing the set of accessible terms; and improve the definition of the accessibility ordering $\succ^X_{acc}$, possibly by making it recursive.

A more challenging problem to be investigated then is the generalization of this new definition to the calculus of constructions along the lines of [51] and the suggestions made in [40], where an RPO-like ordering on types was proposed which allowed to give a single definition for terms and types. Starting this work with definition 1 is of course desirable.

Finally, it appears that the recursive path ordering and the computability 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. 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. An intriguing, challenging question is therefore to understand the precise relationship between computability predicates and path orderings.

Acknowledgements: the second author wishes to point out 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. We are also in debt with Cynthia Kop and Femke van Raamsdonk for pointing out to us a (hopefully minor) mistake in published versions of our work on HORPO.

References

1. A. Abel. Termination checking with types. *Theoretical Informatics and Applications*, 38(4):277–319, 2004.
2. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
3. F. Barbanera. Adding algebraic rewriting to the Calculus of Constructions: strong normalization preserved. In *Proceedings of the 2nd International Workshop on Conditional and Typed Rewriting Systems*, Lecture Notes in Computer Science 516, 1990.
4. F. Barbanera and M. Fernández. Combining first and higher order rewrite systems with type assignment systems. In *Proceedings of the 1st International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 664, 1993.
5. F. Barbanera and M. Fernández. Modularity of termination and confluence in combinations of rewrite systems with λ_ω. In *Proceedings of the 20th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science 700, 1993.
6. F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalization and confluence in the algebraic-λ-cube. In *Proceedings of the 9th IEEE Symposium on Logic in Computer Science*, 1994.
7. G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.
8. A. M. Ben-Amram, N. D. Jones, and C. S. Lee. The size-change principle for program termination. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages, 2001*.
9. F. Blanqui. Definitions by rewriting in the Calculus of Constructions (extended abstract). In *Proceedings of the 16th IEEE Symposium on Logic in Computer Science*, 2001.
10. F. Blanqui. Higher-order dependency pairs. In *Proceedings of the 8th International Workshop on Termination, 2006*.
11. F. Blanqui. Termination and confluence of higher-order rewrite systems. In *Proceedings of the 11th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 1833, 2000.
12. F. Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 3091, 2004.
13. F. Blanqui. Definitions by rewriting in the Calculus of Constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.
14. F. Blanqui. Inductive types in the Calculus of Algebraic Constructions. *Fundamenta Informaticae*, 65(1-2):61–86, 2005.
15. F. Blanqui. Computability closure: Ten years later. In *Rewriting, Computation and Proof – Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, volume 4600 of *Lecture Notes in Computer Science*, 2007.
16. F. Blanqui, J.-P. Jouannaud, and M. Okada. Inductive-data-type Systems. *Theoretical Computer Science*, 272:41–68, 2002.
17. F. Blanqui, J.-P. Jouannaud, and A. Rubio. Higher-order termination: from Kruskal to computability. In *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Lecture Notes in Computer Science 4246, 2006.
18. F. Blanqui, J.-P. Jouannaud, and A. Rubio. HORPO with computability closure : A reconstruction. In *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Lecture Notes in Computer Science 4790, 2007.
19. N. Bohr and N. Jones. Termination Analysis of the untyped lambda-calculus. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 3091, pp 1-23, 2004.

20. C. Borralleras. *Ordering-based methods for proving termination automatically*. PhD thesis, Universitat Politècnica de Catalunya, Spain, 2003.
21. C. Borralleras and A. Rubio. A monotonic higher-order semantic path ordering. In *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Lecture Notes in Computer Science 2250, 2001.
22. C. Borralleras and A. Rubio. Orderings and constraints: Theory and practice of proving termination. In *Rewriting, Computation and Proof – Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, volume 4600 of *Lecture Notes in Computer Science*, 2007.
23. V. Breazu-Tannen. Combining algebra and higher-order types. In *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science*, 1988.
24. V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. In *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science 372, 1989.
25. W. N. Chin and S. C. Khoo. Calculating sized types. *Journal of Higher-Order and Symbolic Computation*, 14(2-3):261–300, 2001.
26. N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.
27. N. Dershowitz. Personal communication, 2008.
28. D. Dougherty. Adding algebraic rewriting to the untyped lambda calculus. *Information and Computation*, 101(2):251–267, 1992.
29. J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated termination analysis for haskell: From term rewriting to programming languages. In *Proceedings of the 17th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 4098, 2006.
30. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Lecture Notes in Computer Science 3452, 2004.
31. J. Goubault-Larrecq. Well-founded recursive relations. In *Proceedings of the 15th International Conference on Computer Science Logic*, Lecture Notes in Computer Science 2142, 2001.
32. J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23th ACM Symposium on Principles of Programming Languages, 1996*.
33. C. B. Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems*, 26(6):911–937, 2004.
34. J.-P. Jouannaud and M. Okada. A computation model for executable higher-order algebraic specification languages. In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, 1991.
35. J.-P. Jouannaud and M. Okada. Abstract Data Type Systems. *Theoretical Computer Science*, 173(2):349–391, 1997.
36. J.-P. Jouannaud and A. Rubio. Higher-order orderings for normal rewriting. In *Proceedings of the 17th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 4098, 2006.
37. J.-P. Jouannaud and A. Rubio. The Higher-Order Recursive Path Ordering. In *Proceedings of the 14th IEEE Symposium on Logic in Computer Science*, 1999.
38. J.-P. Jouannaud and A. Rubio. Rewrite orderings for higher-order terms in eta-long beta-normal form and the recursive path ordering. *Theoretical Computer Science*, 208:33–58, 1998.

39. J.-P. Jouannaud and A. Rubio. Higher-order recursive path orderings *"à la carte"*, 2001. Draft.
40. J.-P. Jouannaud and A. Rubio. Polymorphic higher-order recursive path orderings. *Journal of the ACM*, 54(1):1–48, 2007.
41. S. Kamin and J.-J. Lévy. Two generalizations of the Recursive Path Ordering, 1980. Unpublished.
42. M. S. Krishnamoorthy and P. Narendran. On recursive path ordering. *Theoretical Computer Science*, 40(2-3):323–328, 1985.
43. C. Loria-Saenz and J. Steinbach. Termination of combined (rewrite and λ-calculus) systems. In *Proceedings of the 3rd International Workshop on Conditional and Typed Rewriting Systems*, Lecture Notes in Computer Science 656, 1992.
44. M. Okada. Strong normalizability for the combined system of the typed lambda calculus and an arbitrary convergent term rewrite system. In *Proceedings of the 1989 International Symposium on Symbolic and Algebraic Computation*, ACM Press.
45. M. Sakai and K. Kusakari. On dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E88-D(3):583–593, 2005.
46. M. Sakai, Y. Watanabe, and T. Sakabe. An extension of dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E84-D(8):1025–1032, 2001.
47. J. van de Pol. Termination proofs for higher-order rewrite systems. In *Proceedings of the 1st International Workshop on Higher-Order Algebra, Logic and Term Rewriting*, Lecture Notes in Computer Science 816, 1993.
48. J. van de Pol. *Termination of higher-order rewrite systems*. PhD thesis, Utrecht Universiteit, Nederlands, 1996.
49. J. van de Pol and H. Schwichtenberg. Strict functionals for termination proofs. In *Proceedings of the 2nd International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 902, 1995.
50. F. van Raamsdong and C. Kop. Personal communication, 2008.
51. D. Walukiewicz-Chrzaszcz. Termination of rewriting in the Calculus of Constructions. *Journal of Functional Programming*, 13(2):339–414, 2003.
52. H. Xi. Dependent types for program termination verification. *Journal of Higher-Order and Symbolic Computation*, 15(1):91–131, 2002.