Higher-Order Termination: From Kruskal to Computability

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

[1] INRIA & LORIA, BP 101, 54602 Villiers-lés-Nancy CEDEX, France
[2] LIX, École Polytechnique, 91400 Palaiseau, France
[3] Technical University of Catalonia, Pau Gargallo 5, 08028 Barcelona, Spain

1 Introduction

Termination is a major question in both logic and computer science. In logic, termination is at the heart of proof theory where it is usually called strong normalization (of cut elimination). In computer science, termination has always been an important issue for showing programs correct. In the early days of logic, strong normalization was usually shown by assigning ordinals to expressions in such a way that eliminating a cut would yield an expression with a smaller ordinal. In the early days of verification, computer scientists used similar ideas, interpreting the arguments of a program call by a natural number, such as their size. Showing the size of the arguments to decrease for each recursive call gives a termination proof of the program, which is however rather weak since it can only yield quite small ordinals. In the sixties, Tait invented a new method for showing cut elimination of natural deduction, based on a *predicate* over the set of terms, such that the membership of an expression to the predicate implied the strong normalization property for that expression. The predicate being defined by induction on types, or even as a fixpoint, this method could yield much larger ordinals. Later generalized by Girard under the name of *reducibility* or *computability candidates*, it showed very effective in proving the strong normalization property of typed lambda-calculi with polymorphic types, dependent types, inductive types, and finally a cumulative hierarchy of universes. On the programming side, research on termination shifted from programming to executable specification languages based on rewriting, and concentrated on automatable methods based on the construction on well-founded orderings of the set of terms. The milestone here is Dershowitz's *recursive path ordering* (RPO), in the late seventies, whose well-foundedness proof is based on a powerful combinatorial argument, Kruskal's tree theorem, which also yields rather large ordinals. While the computability predicates must be defined for each particular case, and their properties proved by hand, the recursive path ordering can be effectively automated.

These two methods are completely different. Computability arguments show *termination*, that is, infinite decreasing sequences of expressions $e_0 \succ e_1 \succ \ldots e_n \succ e_{n+1} \ldots$ do not exist. Kruskal's based arguments show *well-orderedness*: for any infinite sequence of expressions $\{e_i\}_i$, there is a pair $j < k$ such that $e_j \preceq e_k$. It is easy to see that well-orderedness implies termination, but the converse is not true.

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

In the late eighties, a new question arose: termination of a simply-typed lambda-calculus language in which beta-reduction would be supplemented with terminating first-order rewrite rules. Breazu-Tannen and Gallier on the one hand [12], and Okada [23] on the other hand, showed that termination was satisfied by the combination by using computability arguments. Indeed, when rewriting operates at basic types and is generated by first-order rewrite rules, beta-reduction and rewriting do not interfere. Their result, proved for a polymorphic λ-calculus, was later generalized to the calculus of constructions [1]. The situation becomes radically different with higher-order rewriting generated by rules operating on arrow-types, or involving lambda-bindings or higher-order variables. Such an example is provided by Gödel's system T, in which higher-order primitive recursion for natural numbers generated by Peano's constructors 0 and s is described by the following two higher-order rules :

$$\begin{aligned} rec(0, U, V) &\to U \\ rec(s(X), U, V) &\to @(V, X, rec(X, U, V)) \end{aligned}$$

where rec is a function symbol of type $\mathbb{N} \to T \to (\mathbb{N} \to T \to T) \to T$, U is a higher-order variable of type T and V a higher-order variable of type $\mathbb{N} \to T \to T$, for all type T. Jouannaud and Okada invented the so-called general-schema [17], a powerful generalization of Gödel's higher-order primitive recursion of higher types. Following the path initiated by Breazu-Tannen and Gallier on the one hand, and Okada on the other hand, termination of calculi based on the general schema was proved by using computability arguments as well [17,18,2]. The general schema was then reformulated by Blanqui, Jouannaud and Okada [3,4] in order to incorporate computability arguments directly in its definition, opening the way to new generalizations. Gödel's system T can be generalized in two ways, by introducing type constructors and dependent types, yielding the Calculus of Constructions, and by introducing strictly positive inductive types. Both together yield the Calculus of Inductive Constructions [24], the theory underlying the Coq system [14], in which rewrite rules like strong elimination operate on types, raising new difficulties. Blanqui gave a generalization of the general schema which includes the Calculus of Inductive Constructions as a particular case under the name of Calculus of Algebraic Constructions [6,7].

The general schema, however, is too simple to analyze complex calculi defined by higher-order rewrite rules such as encodings of logics. For that purpose, Jouannaud and Rubio generalized the recursive path ordering to the higher-order case, yielding the higher-order recursive path ordering (HORPO) [19]. The RPO well-foundedness proof follows from Kruskal's tree theorem, but no such theorem exists in presence of a binding construct, and it is not at all clear that such a theorem may exist. What is remarkable is that computability arguments fit with RPO's recursive structure. When applied to RPO, these arguments result in a new, simple, well-foundedness proof of RPO. One could even argue that this is the *first* well-foundedness proof of RPO, since Dershowitz showed *more*: well-orderedness.

Combining the general schema and the HORPO is indeed easy because their termination properties are both based on computability arguments. The resulting relation, HORPO with closure, combines an ordering relation with a membership predicate. In this paper, we reformulate and improve a recent idea of Blanqui [9] by defining a new

version of the HORPO with closure which integrates smoothly the idea of the general schema into HORPO in the form of a new ordering definition.

So far, we have considered the kind of higher-order rewriting defined by using first-order pattern matching as in the calculus of constructions. These orderings need to contain β- and η-reductions. Showing termination of higher-order rewrite rules based on higher-order pattern matching, that is, rewriting modulo β and η now used as equalities, turns out to require simple modifications of HORPO [20]. We will therefore concentrate here on higher-order orderings containing β- and η-reductions.

We introduce higher-order algebras in Section 2. In Section 3, we recall the computability argument for this variation of the simply typed lambda calculus. Using a computability argument again, we show in Section 4 that RPO is well-founded. We introduce the general schema in section 5, and the HORPO in Section 6 before to combine both in Section 7. We end up with related work and open problems in the last two sections.

2 Higher-Order Algebras

The notion of a higher-order algebra given here is the monomorphic version of the notion of polymorphic higher-order algebra defined in [21]. Polymorphism has been ruled out for simplicity.

2.1 Types, Signatures and Terms

Given a set $\mathcal{S}$ of *sort symbols* of a fixed arity, denoted by $s : *^n \Rightarrow *$, the set $\mathcal{T}_\mathcal{S}$ of *types* is generated from these sets by the arrow constructor:

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

Types headed by $\to$ are *arrow types* while the others are *basic types*. *Type declarations* are expressions of the form $\sigma_1 \times \cdots \times \sigma_n \to \sigma$, where n is the *arity* of the type declaration, and $\sigma_1, \ldots, \sigma_n, \sigma$ are types. A type declaration is *first-order* if it uses only sorts, otherwise *higher-order*.

We assume given a set of function symbols which are meant to be algebraic operators. Each function symbol f is equipped with a type declaration $f\colon \sigma_1 \times \cdots \times \sigma_n \to \sigma$. We use $\mathcal{F}_n$ for the set of function symbols of arity n. $\mathcal{F}$ is a *first-order signature* if all its type declarations are first-order, and a higher-order signature otherwise.

The set of *raw terms* is generated from the signature $\mathcal{F}$ and a denumerable set $\mathcal{X}$ of variables according to the grammar:

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

Terms generated by the first two grammar rules are called *algebraic*. Terms of the form $\lambda x.u$ are called *abstractions* while terms of the form $@(u, v)$ are called *applications*. The term $@(u, \overline{v})$ is called a (partial) *left-flattening* of $@(\ldots @(@(u, v_1), v_2), \ldots, v_n)$, with u being possibly an application itself. Terms other than abstractions are said to be

neutral. We denote by $\mathcal{V}ar(t)$ ($\mathcal{BV}ar(t)$) the set of free (bound) variables of t. We may assume for convenience (and without further notice) that bound variables in a term are all different, and are different from the free ones.

Terms are identified with finite labeled trees by considering $\lambda x.$, for each variable x, as a unary function symbol. *Positions* are strings of positive integers, the empty string Λ denoting the root position. The *subterm* of t at position p is denoted by $t|_p$, and by $t[u]_p$ the result of replacing $t|_p$ at position p in t by u. We write $s \rhd u$ if u is a strict subterm of s. We use $t[\,]_p$ for a term with a hole, called a context. The notation $\overline{s}$ will be ambiguously used to denote a list, a multiset, or a set of terms $s_1, \ldots, s_n$.

2.2 Typing Rules

Typing rules restrict the set of terms by constraining them to follow a precise discipline. Environments are sets of pairs written $x : \sigma$, where x is a variable and σ is a type. Let $\mathcal{D}om(\Gamma) = \{x \mid x : \sigma \in \Gamma \text{ for some type } \sigma\}$. We assume there is a unique pair of the form $x : \sigma$ for every variable $x \in \mathcal{D}om(\Gamma)$. Our typing judgments are written as $\Gamma \vdash M : \sigma$ if the term M can be proved to have the type σ in the environment Γ. A term M has type σ in the environment Γ if $\Gamma \vdash M : \sigma$ is provable in the inference system of Figure 1. A term M is typable in the environment Γ if there exists a type σ such that M has type σ in the environment Γ. A term M is typable if it is typable in some environment Γ. Note that function symbols are uncurried, hence must come along with all their arguments.

Variables:
$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

Functions:
$$\frac{f : \sigma_1 \times \ldots \times \sigma_n \to \sigma \quad \Gamma \vdash t_1 : \sigma_1 \ \ldots \ \Gamma \vdash t_n : \sigma_n}{\Gamma \vdash f(t_1, \ldots, t_n) : \sigma}$$

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

Application:
$$\frac{\Gamma \cup \{x : \sigma\} \vdash s : \sigma \to \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash @(s, t) : \tau}$$

Fig. 1. Typing judgments in higher-order algebras

2.3 Higher-Order Rewrite Rules

Substitutions are written as in $\{x_1 : \sigma_1 \mapsto (\Gamma_1, t_1), \ldots, x_n : \sigma_n \mapsto (\Gamma_n, t_n)\}$ where, for every $i \in [1..n]$, t_i is assumed different from x_i and $\Gamma_i \vdash t_i : \sigma_i$. We also assume that $\bigcup_i \Gamma_i$ is an environment. We often write $x \mapsto t$ instead of $x : \sigma \mapsto (\Gamma, t)$, in particular when t is ground. We use the letter γ for substitutions and postfix notation for their application. Substitutions behave as endomorphisms defined on free variables.

A (possibly higher-order) *term rewriting system* is a set of rewrite rules $R = \{\Gamma_i \vdash l_i \to r_i : \sigma_i\}_i$, where l_i and r_i are higher-order terms such that l_i and r_i have the same type σ_i in the environment Γ_i. Given a term rewriting system R, a term s rewrites to a term t at position p with the rule $l \to r$ and the substitution γ, written $s \xrightarrow[l \to r]{p} t$, or simply $s \to_R t$, if $s|_p = l\gamma$ and $t = s[r\gamma]_p$.

A term s such that $s \xrightarrow[R]{p} t$ is called *R-reducible*. The subterm $s|_p$ is a *redex* in s, and t is the *reduct* of s. Irreducible terms are said to be in *R-normal form*. A substitution γ is in R-normal form if $x\gamma$ is in R-normal form for all x. We denote by $\xrightarrow[R]{*}$ the reflexive, transitive closure of the rewrite relation $\xrightarrow[R]{}$.

Given a rewrite relation $\longrightarrow$, a term s is strongly normalizing if there is no infinite sequence of rewrites issuing from s. The rewrite relation itself is *strongly normalizing*, or *terminating*, if all terms are strongly normalizing, in which case it is called a *reduction*.

Three particular higher-order equation schemas originate from the λ-calculus, α-, β- and η-equality:

$$\begin{array}{rll} \lambda x.v & =_\alpha & \lambda y.v\{x \mapsto y\} \text{ if } y \notin \mathcal{BV}ar(v) \cup (\mathcal{V}ar(v) \setminus \{x\}) \\ @(\lambda x.v, u) & \longrightarrow_\beta & v\{x \mapsto u\} \\ \lambda x.@(u, x) & \longrightarrow_\eta & u \qquad\qquad \text{if } x \notin \mathcal{V}ar(u) \end{array}$$

As usual, we do not distinguish α-convertible terms. β- and η-equalities are used as reductions, which is indicated by the long-arrow symbol instead of the equality symbol. The above rule-schemas define a rewrite system which is known to be terminating, a result proved in Section 3.

2.4 Higher-Order Reduction Orderings

We will make intensive use of well-founded orderings, using the vocabulary of rewrite systems for orderings, for proving strong normalization properties. For our purpose, an *ordering*, usually denoted by $\geq$, is a reflexive, symmetric, transitive relation compatible with α-conversion, that is, $s =_\alpha t \geq u =_\alpha v$ implies $s \geq v$, whose strict part $>$ is itself compatible. We will essentially use strict orderings, and hence, the word ordering for them too. We will also make use of order-preserving operations on relations, namely multiset and lexicographic extensions, see [15].

Rewrite orderings are *monotonic* and *stable* orderings, *reduction orderings* are in addition *well-founded*, while *higher-order reduction orderings* must also contain β- and η-reductions. Monotonicity of $>$ is defined as $u > v$ implies $s[u]_p > s[v]_p$ for all contexts $s[\,]_p$. Stability of $>$ is defined as $u > v$ implies $s\gamma > t\gamma$ for all substitutions γ. Higher-order reduction orderings are used to prove termination of rewrite systems including β- and η-reductions by simply comparing the left hand and right hand sides of the remaining rules.

3 Computability

Simply minded arguments do not work for showing the strong normalization property of the simply typed lambda-calculus, for β-reduction increases the size of terms, which

precludes an induction on their size, and preserves their types, which seems to preclude an induction on types.

Tait's idea is to generalize the strong normalization property in order to enable an induction on types. To each type σ, we associate a subset $[\![\sigma]\!]$ of the set of terms, called the *computability predicate* of type σ, or set of *computable terms* of type σ. Whether $[\![\sigma]\!]$ contains only typable terms of type σ is not really important, although it can help intuition. What is essential are the properties that the family of predicates should satisfy:

(i) computable terms are strongly normalizing;
(ii) reducts of computable terms are computable;
(iii) a neutral term u is computable iff all its reducts are computable;
(iv) $u : \sigma \to \tau$ is computable iff so is $@(u, v)$ for all computable v.

A (non-trivial) consequence of all these properties can be added to smooth the proof of the coming Main Lemma:

(v) $\lambda x.u$ is computable iff so is $u\{x \mapsto v\}$ for all computable v.

Apart from (v), the above properties refer to β-reduction via the notions of *reduct* and *strong normalization* only. Indeed, various computability predicates found in the literature use the same definition parametrized by the considered reduction relation.

There are several ways to define a computability predicate by taking as its definition some of the properties that it should satisfy. For example, a simple definition by induction on types is this:

$s : \sigma \in [\![\sigma]\!]$ for σ basic iff s is strongly normalizing;
$s : \theta \to \tau \in [\![\sigma \to \tau]\!]$ iff $@(s, u) : \tau \in [\![\tau]\!]$ for every $u : \theta \in [\![\theta]\!]$.

An alternative for the case of basic type is:

$$s : \sigma \in [\![\sigma]\!] \text{ iff } \forall t : \tau \,.\, s \longrightarrow t \text{ then } t \in [\![\tau]\!].$$

This formulation defines the predicate as a fixpoint of a monotonic functional. Once the predicate is defined, it becomes necessary to show the computability properties. This uses an induction on types in the first case or an induction on the definition of the predicate in the fixpoint case.

Tait's strong normalization proof is based on the following key lemma:

Lemma 1 (Main Lemma). *Let s be an arbitrary term and γ be an arbitrary computable substitution. Then $s\gamma$ is computable.*

Proof. By induction on the structure of terms.

1. s is a variable: $s\gamma$ is computable by assumption on γ.
2. $s = @(u, v)$. Since $u\gamma$ and $v\gamma$ are computable by induction hypothesis, $s\gamma = @(u\gamma, v\gamma)$ is computable by computability property (iv).
3. $s = \lambda x.u$. By computability property (v), $s\gamma = \lambda x.u\gamma$ is computable iff $u\gamma\{x \mapsto v\}$ is computable for an arbitrary computable v. Let now $\gamma' = \gamma \cup \{x \mapsto v\}$. By definition of substitutions for abstractions, $u\gamma\{x \mapsto v\} = u\gamma'$, which is usually ensured by α-conversion. By assumptions on γ and v, γ' is computable, and $u\gamma'$ is therefore computable by the main induction hypothesis. □

Since an arbitrary term s can be seen as its own instance by the identity substitution, which is computable by computability property (iii), all terms are computable by the Main Lemma, hence strongly normalizing by computability property (i).

4 The Recursive Path Ordering and Computability

In this section, we restrict ourselves to first-order algebraic terms. Assuming that the set of function symbols is equipped with an ordering relation $\geq_{\mathcal{F}}$, called *precedence*, and a status function $stat$, writing $stat_f$ for $stat(f)$, we recall the definition of the recursive path ordering:

Definition 1. $s \succ_{rpo} t$ *iff*

1. $s = f(\overline{s})$ *with* $f \in \mathcal{F}$, *and* $u \underset{rpo}{\succeq} t$ *for some* $u \in \overline{s}$
2. $s = f(\overline{s})$ *with* $f \in \mathcal{F}$, *and* $t = g(\overline{t})$ *with* $f >_{\mathcal{F}} g$, *and* A
3. $s = f(\overline{s})$ *and* $t = g(\overline{t})$ *with* $f =_{\mathcal{F}} g \in Mul$, *and* $\overline{s}\ (\underset{rpo}{\succ})_{mul}\ \overline{t}$
4. $s = f(\overline{s})$ *and* $t = g(\overline{t})$ *with* $f =_{\mathcal{F}} g \in Lex$, *and* $\overline{s}\ (\underset{rpo}{\succ})_{lex}\ \overline{t}$ *and* A

$$\textit{where } A = \forall v \in \overline{t}.\ s \underset{rpo}{\succ} v \quad \textit{and} \quad s \underset{rpo}{\succeq} t \textit{ iff } s \underset{rpo}{\succ} t \textit{ or } s = t$$

We now show the well-foundedness of $\succ_{rpo}$ by using Tait's method. Computability is defined here as strong normalization, implying computability property (i). We prove the computability property:

(vii) Let $f \in \mathcal{F}_n$ and $\overline{s}$ be computable terms. Then $f(\overline{s})$ is computable.

Proof. The restriction of $\succ_{rpo}$ to terms smaller than or equal to the terms in $\overline{s}$ w.r.t. $\succ_{rpo}$ is a well-founded ordering which we use for building an outer induction on the pairs $(f, \overline{s})$ ordered by $(>_{\mathcal{F}}, (\succ_{rpo})_{stat_f})_{lex}$. This ordering is well-founded, since it is built from well-founded orderings by using mappings that preserve well-founded orderings.

We now show that $f(\overline{s})$ is computable by proving that t is computable for all t such that $f(\overline{s}) \succ_{rpo} t$. This property is itself proved by an (inner) induction on $|t|$, and by case analysis upon the proof that $f(\overline{s}) \succ_{rpo} t$.

1. subterm case: $\exists u \in \overline{s}$ such that $u \succ_{rpo} t$. By assumption, u is computable, hence so is its reduct t.
2. precedence case: $t = g(\overline{t})$, $f >_{\mathcal{F}} g$, and $\forall v \in \overline{t}$, $s \succ_{rpo} v$. By inner induction, v is computable, hence so is $\overline{t}$. By outer induction, $g(\overline{t}) = t$ is computable.
3. multiset case: $t = f(\overline{t})$ with $f \in Mul$, and $\overline{s}(\succ_{rpo})_{mul}\overline{t}$. By definition of the multiset extension, $\forall v \in \overline{t},\ \exists u \in \overline{s}$ such that $u \succeq_{rpo} v$. Since $\overline{s}$ is a vector of computable terms by assumption, so is $\overline{t}$. We conclude by outer induction that $f(\overline{t}) = t$ is computable.
4. lexicographic case: $t = f(\overline{t})$ with $f \in Lex$, $\overline{s}(\succ_{rpo})_{lex}\overline{t}$, and $\forall v \in \overline{t},\ s \succ_{rpo} v$. By inner induction, $\overline{t}$ is strongly normalizing, and by outer induction, so is $f(\overline{t}) = t$.

□

The well-foundedness of $\succ_{rpo}$ follows from computability property (vii).

5 The General Schema and Computability

As in the previous section, we assume that the set of function symbols is equipped with a precedence relation $\geq_{\mathcal{F}}$ and a status function $stat$.

Definition 2. *The* computability closure $\mathcal{CC}(t = f(\overline{t}))$*, with* $f \in \mathcal{F}$*, is the set* $\mathcal{CC}(t, \emptyset)$*, s.t.* $\mathcal{CC}(t, \mathcal{V})$*, with* $\mathcal{V} \cap \mathcal{V}ar(t) = \emptyset$*, is the smallest set of typable terms containing all variables in* $\mathcal{V}$ *and terms in* $\overline{t}$*, closed under:*

1. *subterm of basic type: let* $s \in \mathcal{CC}(t, \mathcal{V})$*, and* u *be a subterm of* s *of basic type* σ *such that* $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$*; then* $u \in \mathcal{CC}(t, \mathcal{V})$*;*
2. *precedence: let* $f >_{\mathcal{F}} g$*, and* $\overline{s} \in \mathcal{CC}(t, \mathcal{V})$*; then* $g(\overline{s}) \in \mathcal{CC}(t, \mathcal{V})$*;*
3. *recursive call: let* $f(\overline{s})$ *be a term such that terms in* $\overline{s}$ *belong to* $\mathcal{CC}(t, \mathcal{V})$ *and* $\overline{t}(\longrightarrow_{\beta} \cup \rhd)_{stat_f} \overline{s}$*; then* $g(\overline{s}) \in \mathcal{CC}(t, \mathcal{V})$ *for every* $g =_{\mathcal{F}} f$*;*
4. *application: let* $s : \sigma_1 \to \ldots \to \sigma_n \to \sigma \in \mathcal{CC}(t, \mathcal{V})$ *and* $u_i : \sigma_i \in \mathcal{CC}(t, \mathcal{V})$ *for every* $i \in [1..n]$*; then* $@(s, u_1, \ldots, u_n) \in \mathcal{CC}(t, \mathcal{V})$*;*
5. *abstraction: let* $s \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$ *for some* $x \notin \mathcal{V}ar(t) \cup \mathcal{V}$*; then* $\lambda x.s \in \mathcal{CC}(t, \mathcal{V})$*;*
6. *reduction: let* $u \in \mathcal{CC}(t, \mathcal{V})$*, and* $u \longrightarrow_{\beta \cup \rhd} v$*; then* $v \in \mathcal{CC}(t, \mathcal{V})$*.*

We say that a rewrite system R satisfies the *general schema* iff

$$r \in \mathcal{CC}(f(\overline{l})) \text{ for all } f(\overline{l}) \to r \in R$$

We now consider computability with respect to the rewrite relation $\longrightarrow_R \cup \longrightarrow_{\beta}$, and add the computability property (vii) whose proof can be easily adapted from the previous one. We can then add a new case in Tait's Main Lemma, for terms headed by an algebraic function symbol. As a consequence, the relation $\longrightarrow_{\beta} \cup \longrightarrow_R$ is strongly normalizing.

Example 1 (System T). We show the strong normalization of Gödel's system T by showing that its rules satisfy the general schema. This is clear for the first rule by the base Case of the definition. For the second rule, we have: $V \in \mathcal{CC}(rec(s(X), U, V))$ by base Case; $s(X) \in \mathcal{CC}(rec(s(X), U, V))$ by base Case again, and $X \in \mathcal{CC}(rec(s(X), U, V))$ by Case 2, assuming $rec >_{\mathcal{F}} s$; $U \in \mathcal{CC}(rec(s(X), U, V))$ by base Case, hence all arguments of the recursive call are in $\mathcal{CC}(rec(s(X), U, V))$. Since $s(X) \rhd X$ holds, we have $rec(X, U, V) \in \mathcal{CC}(rec(s(X), U, V))$. Therefore, we conclude with $@(V, X, rec(X, U, V)) \in \mathcal{CC}(rec(s(X), U, V)$ by Case 4.

6 The Higher-Order Recursive Path Ordering

6.1 The Ingredients

– A quasi-ordering on types $\geq_{\mathcal{TS}}$ called *the type ordering* satisfying the following properties:
  1. *Well-foundedness*: $>_{\mathcal{TS}}$ is well-founded;
  2. *Arrow preservation*: $\tau \to \sigma =_{\mathcal{TS}} \alpha$ iff $\alpha = \tau' \to \sigma'$, $\tau' =_{\mathcal{TS}} \tau$ and $\sigma =_{\mathcal{TS}} \sigma'$;
  3. *Arrow decreasingness*: $\tau \to \sigma >_{\mathcal{TS}} \alpha$ implies $\sigma \geq_{\mathcal{TS}} \alpha$ or $\alpha = \tau' \to \sigma', \tau' =_{\mathcal{TS}} \tau$ and $\sigma >_{\mathcal{TS}} \sigma'$;

4. *Arrow monotonicity*: $\tau \geq_{\mathcal{T}_S} \sigma$ implies $\alpha \to \tau \geq_{\mathcal{T}_S} \alpha \to \sigma$ and $\tau \to \alpha \geq_{\mathcal{T}_S} \sigma \to \alpha$;

A convenient type ordering is obtained by restricting the subterm property for the arrow in the RPO definition.

- A quasi-ordering $\geq_{\mathcal{F}}$ on $\mathcal{F}$, called the *precedence*, such that $>_{\mathcal{F}}$ is well-founded.
- A *status* $stat_f \in \{Mul, Lex\}$ for every symbol $f \in \mathcal{F}$.

The higher-order recursive path ordering (HORPO) operates on typing judgments. To ease the reading, we will however forget the environment and type unless necessary. Let

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

Definition 3. *Given two judgments $\Gamma \vdash_\Sigma s : \sigma$ and $\Sigma \vdash_\Sigma t : \tau$,*

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

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

Example 2 (System T). The new proof of strong normalization of System T is even simpler. For the first rule, we apply Case 1. For the second, we apply Case 7, and show recursively that $rec(s(X), U, V) \succ_{horpo} V$ by Case 1, $rec(s(X), U, V) \succ_{horpo} X$ by Case 1 applied twice, and $rec(s(X), U, V) \succ_{horpo} rec(X, U, V)$ by Case 3, assuming a multiset status for rec, which follows from the comparison $s(X) \succ_{horpo} X$ by Case 1.

The strong normalization proof of HORPO is in the same style as the previous strong normalization proofs, although technically more complex [21]. This proof shows that HORPO and the general schema can be combined by replacing the membership $u \in \overline{s}$ used in case 1 by the more general membership $u \in \mathcal{CC}(f(\overline{s}))$. It follows that the HORPO mechanism is inherently more expressive than the closure mechanism.

Because of Cases 11 and 12, HORPO is not transitive. Indeed, there are examples for which the proof of $s \succ^+_{horpo} t$ requires guessing a middle term u such that $s \succ_{horpo} u$

and $u \succ_{horpo} t$. Guessing a middle term when necessary is automated in the implementations of HORPO and HORPO with closure available from the web page of the first two authors.

7 Unifying HORPO and the Computability Closure

A major advantage of HORPO over the general schema is its recursive structure. In contrast, the membership to the computability closure is undecidable due to its Case 3, but does not involve any type comparison. To combine the advantages of both, we now incorporate the closure construction into the HORPO as an ordering. Besides, we also incorporate the property that arguments of a type constructor are computable when the *positivity condition* is satisfied as it is the case for inductive types in the Calculus of Inductive Constructions [24,7].

$$s : \sigma \underset{horpo}{\succ} t : \tau \quad \text{iff} \quad \mathcal{V}ar(t) \subseteq \mathcal{V}ar(s) \quad \text{and}$$

1. $s = f(\overline{s})$ and $s \overset{\emptyset}{\underset{comp}{\succ}} t$
2. $s = f(\overline{s})$ and $\sigma \geq_{\mathcal{TS}} \tau$ and
   (a) $t = g(\overline{t}),\ f >_{\mathcal{F}} g$ and A
   (b) $t = g(\overline{t}),\ f =_{\mathcal{F}} g$, $\overline{s}(\underset{horpo}{\succ})_{stat_f} \overline{t}$ and A
   (c) $t = @(t_1, t_2)$ and A
3. $s = @(s_1, s_2), \sigma \geq_{\mathcal{TS}} \tau$ and
   (a) $t = @(t_1, t_2)$ and $\{s_1, s_2\}(\underset{horpo}{\succ})_{mul}\{t_1, t_2\}$
   (b) $s_1 \underset{horpo}{\succeq} t$ or $s_2 \underset{horpo}{\succeq} t$
   (c) $s_1 = \lambda x.u$ and $u\{x \mapsto s_2\} \underset{horpo}{\succeq} t$
4. $s = \lambda x : \alpha.u, \sigma \geq_{\mathcal{TS}} \tau$ and
   (a) $t = \lambda x : \beta.v, \alpha =_{\mathcal{TS}} \beta$ and $u \underset{horpo}{\succ} v$
   (b) $x \notin \mathcal{V}ar(t)$ and $u \underset{horpo}{\succeq} t$
   (c) $u = @(v, x), x \notin \mathcal{V}ar(v)$ and $v \underset{horpo}{\succeq} t$

where $A = \forall v \in \overline{t} :$
$s \underset{horpo}{\succ} v$ or $\exists u \in \overline{s} :\ u \underset{horpo}{\succ} v$

$$s = f(\overline{s}) \overset{\overline{X}}{\underset{comp}{\succ}} t \quad \text{iff}$$

1. $t \in \overline{X}$
2. $\exists s_i \in Acc(s) :\ s_i \succeq^{\overline{X}}_{comp} t$
3. $t = g(\overline{t}),\ f >_{\mathcal{TS}} g$ and $\forall v \in \overline{t} :\ s \overset{\overline{X}}{\underset{comp}{\succ}} v$
4. $t = g(\overline{t}),\ f =_{\mathcal{TS}} g$, $\forall v \in \overline{t}\ : s \overset{\overline{X}}{\underset{comp}{\succ}} v$ and $Acc(s)(\underset{horpo}{\succ})_{stat_f} \lambda\overline{X}.\overline{t}$
5. $t = @(u, v)$, $s \overset{\overline{X}}{\underset{comp}{\succ}} u$ and $s \overset{\overline{X}}{\underset{comp}{\succ}} v$
6. $t = \lambda x : \alpha.u$ and $s \overset{\overline{X}\cdot\{x:\alpha\}}{\underset{comp}{\succ}} u$

where $s_i \in Acc(f(\overline{s}))$
(s_i is accessible in s)
iff

1. s is the left hand side of an ancestor goal $s \succ_{horpo} u$
2. s is the left hand side of the current goal $s \succ_{comp} u$, and, either $f : \overline{\sigma} \to \sigma$ and σ occurs only positively in σ_i.

Example 3. We consider now the type of Brouwer's ordinals defined from the type $\mathbb{N}$ by the equation $Ord = 0 \uplus s(Ord) \uplus lim(\mathbb{N} \to Ord)$. Note that Ord occurs positively

in the type $\mathbb{N} \to Ord$, and that $\mathbb{N}$ must be smaller or equal to Ord. The recursor for the type Ord is defined as:

$$\begin{aligned} rec(0, U, V, W) &\to U \\ rec(s(X), U, V, W) &\to @(V, X, rec(X, U, V, W)) \\ rec(lim(F), U, V, W) &\to @(W, F, \lambda n.rec(@(F, n), U, V, W)) \end{aligned}$$

We skip the first two rules and concentrate on the third:

1. $rec(lim(F), U, V, W) \succ_{horpo} @(W, F, \lambda n.rec(@(F, n), U, V, W))$
   which, by Case 1 of $\succ_{horpo}$ is replaced by the new goal:
2. $rec(lim(F), U, V, W) \succ^{\emptyset}_{comp} @(W, F, \lambda n.rec(@(F, n), U, V, W))$
   By Case 5 of $\succ_{comp}$, these three goals become:
3. $rec(lim(F), U, V, W) \succ^{\emptyset}_{comp} W$
4. $rec(lim(F), U, V, W) \succ^{\emptyset}_{comp} F$
5. $rec(lim(F), U, V, W) \succ^{\emptyset}_{comp} \lambda n.rec(@(F, n), U, V, W)$
   Since $rec(lim(F), U, V, W)$ originates from Goal 1,
   Goal 3 disappears by Case 2, while Goal 4 becomes:
6. $lim(F) \succ^{\emptyset}_{comp} F$
   which disappears by the same Case since F is accessible in $lim(F)$.
   thanks to the positivity condition. By Case 6, Goal 5 becomes:
7. $rec(lim(F), U, V, W) \succ^{\{n\}}_{comp} rec(@(F, n), U, V, W)$
   Case 4 applies with a lexicographic status for rec, yielding 5 goals:
8. $rec(lim(F), U, V, W) \succ^{\{n\}}_{comp} @(F, n)$
9. $rec(lim(F), U, V, W) \succ^{\{n\}}_{comp} U$
10. $rec(lim(F), U, V, W) \succ^{\{n\}}_{comp} V$
11. $rec(lim(F), U, V, W) \succ^{\{n\}}_{comp} W$
12. $\{lim(F), U, V, W\}(\succ_{horpo})_{lex}\{\lambda n.@(F, n), \lambda n.U, \lambda n.V, \lambda n.W\}$
    Goals 9, 10, 11 disappear by Case 2, while, by Case 5
    Goal 8 generates (a variation of) the solved Goal 4 and the new sub-goal:
13. $rec(lim(F), U, V, W) \succ^{\{n\}}_{comp} n$
    which disappears by Case 1. We are left with Goal 12, which reduces to:
14. $lim(F) \succ_{horpo} \lambda n.@(F, n)$
    which, by Case 1 of $\succ_{horpo}$, then 6 and 5 of $\succ_{comp}$ yields successively:
15. $lim(F) \succ^{\emptyset}_{comp} \lambda n.@(F, n)$
16. $lim(F) \succ^{\{n\}}_{comp} @(F, n)$
    which, by Case 5, generates (a variation of) the Goal 6 and the last goal:
17. $lim(F) \succ^{\{n\}}_{comp} n$
    which succeeds by Case 1, ending the computation.

To show the strong normalization property of this new definition of $\succ_{horpo}$, we need a more sophisticated predicate combining the predicates used for showing the strong normalization of HORPO [21] and CAC [6]. We have not done any proof yet, but we believe that it is well-founded.

It is worth noting that the ordering $\succ_{horpo}$ defined here is in one way less powerful than the one defined in Section 6 using the closure definition of Section 5 because it does not accumulate computable terms for later use anymore. Instead, it deconstructs its left hand side as usual with rpo, and remembers very few computable terms: the accessible ones only. On the other hand, it is more powerful since the recursive case 4 of the closure uses now the full power of $\succ_{horpo}$ for its last comparison instead of simply β-reduction (see [21]). Besides, there is no more type comparison in Case 1 of the definition of $\succ_{horpo}$, a key improvement which remains to be justified formally.

8 Related Work

Termination of higher-order calculi has recently attracted quite a lot of attention. The area is building up, and mostly, although not entirely, based on reducibility techniques.

The case of conditional rewriting has been recently investigated by Blanqui [8]. His results are presented in this conference.

Giesl's dependency pairs method has been generalized to higher-order calculi by using reducibility techniques as described here [25,10]. The potential of this line of work is probably important, but more work in this direction is needed to support this claim.

Giesl [22] has achieved impressive progress for the case of combinator based calculi, such as Haskell programs, by transforming all definitions into a first-order framework, and then proving termination by using first-order tools. Such transformations do not accept explicit binding constructs, and therefore, do not apply to rich λ-calculi such as those considered here. On the other hand, the relationship of these results with computability deserves investigation.

An original, interesting work is Jones's analysis of the flux of redexes in pure lambda-calculus [16], and its use for proving termination properties of functional programs. Whether this method can yield a direct proof of finite developments in pure λ-calculus should be investigated. We also believe that his method can be incorporated to the HORPO by using an interpretation on terms instead of a type comparison, as mentioned in Conclusion.

Byron Cook, Andreas Podelski and Andrey Ribalchenko [13] have developed a quite different and impressive method based on abstract interpretations to show termination of large imperative programs. Their claim is that large programs are more likely to be shown terminating by approximating them before to make an analysis. Note that the use of a well-founded ordering can be seen as a particular analysis. Although impressive, this work is indeed quite far from our objectives.

9 Conclusion

We give here a list of open problems which we consider important. We are ourselves working on some of these. The higher-order recursive path ordering should be seen as a firm step to undergo further developments in different directions, some of which are listed below.

– Two of them have been investigated in the first order framework: the case of associative commutative operators, and the use of interpretations as a sort of elaborated precedence operating on function symbols. The first extension has been carried out for the general schema [5], and the second for a weak form of HORPO [11]. Both should have an important impact for applications, hence deserve immediate attention.
– Enriching the type system with dependent types, a problem considered by Wałukiewicz [26] for the original version of HORPO in which types were compared by a congruence. Replacing the congruence by HORPO recursively called on types as done in [21] for a simpler type discipline raises technical difficulties. The ultimate goal here is to generalize the most recent versions of the ordering including the present one, for applications to the Calculus of Inductive Constructions.
– HORPO does not contain and is not a well-order for the subterm relationship. However, its definition shows that it satisfies a weak subterm property, namely property A. It would be theoretically interesting to investigate whether some Kruskal-like theorem holds for higher-order terms with respect to the weak subterm property. This could yield an alternative, more abstract way of hiding away computability arguments.

References

1. F. Barbanera. Adding algebraic rewriting to the calculus of constructions: Strong normalization preserved. In *Proc. 2nd Int. Workshop on Conditional and Typed Rewriting Systems, Montreal, LNCS 516*, 1990.
2. F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalization and confluence in the λ-algebraic-cube. In *Proc. 9th IEEE Symp. Logic in Computer Science*, pages 406–415, 1994.
3. F. Blanqui, J.-P. Jouannaud, and M. Okada. The Calculus of Algebraic Constructions. In Paliath Narendran and Michael Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer Verlag.
4. F. Blanqui, J.-P. Jouannaud, and M. Okada. Inductive-Data-Type Systems. *Theoretical Computer Science*, 272(1-2):41–68, 2002.
5. F. Blanqui. Rewriting modulo in Deduction modulo. In *Proc. of the 14th International Conference on Rewriting Techniques and Applications*, volume 2706 of *LNCS*, 2003.
6. F. Blanqui. Definitions by rewriting in the Calculus of Constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.
7. F. Blanqui. Inductive types in the Calculus of Algebraic Constructions. *Fundamenta Informaticae*, 65(1-2):61–86, 2005.
8. F. Blanqui and C. Riba. Combining typing and size constraints for checking termination of higher-order conditional rewriting systems. In *Proc. LPAR, to appear in LNAI*, LNCS, 2006.
9. F. Blanqui. (HO)RPO revisited, 2006. Manuscript.
10. F. Blanqui. Higher-order dependency pairs. In *Proceedings of the 8th International Workshop on Termination, 2006*.
11. Cristina Borralleras and Albert Rubio. A monotonic, higher-order semantic path ordering. In *Proceedings LPAR*, Lecture Notes in Computer Science. Springer Verlag, 2006.
12. Val Breazu-Tannen and Jean Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 1990.

13. Andreas Podelski Byron Cook and Andrey Rybalchenko. Termination proofs for systems code, 2004. Manuscript.
14. Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.0*. INRIA Rocquencourt, France, 2004. `http://coq.inria.fr/`.
15. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–309. North Holland, 1990.
16. Neil Jones and Nina Bohr. Termination analysis of the untyped λ-calculus. In *Rewriting techniques and Applications*, pages 1–23. Springer Verlag, 2004. LNCS 3091.
17. Jean-Pierre Jouannaud and Mitsuhiro Okada. Executable higher-order algebraic specification languages. In *Proc. 6th IEEE Symp. Logic in Computer Science, Amsterdam*, pages 350–361, 1991.
18. Jean-Pierre Jouannaud and Mitsuhiro Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349–391, February 1997.
19. 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.
20. Jean-Pierre Jouannaud and Albert Rubio. Higher-order orderings for normal rewriting. In *Proc. 17th International Conference on Rewriting Techniques and Applications, Seattle, Washington, USA*, 2006.
21. Jean-Pierre Jouannaud and Albert Rubio. Polymorphic higher-order recursive path orderings. *Journal of the ACM*, submitted.
22. Peter Scneider-Kamp Jürgen Giesl, Stephan Swiderski and René Thiemann. Automated termination analysis for haskell: Form term rewriting to programming languages. In *Rewriting techniques and Applications*, pages 297–312. Springer Verlag, 2006. LNCS 4098.
23. Mitsuhiro Okada. Strong normalizability for the combined system of the typed lambda calculus and an arbitrary convergent term rewrite system. In *Proc. of the 20th Int. Symp. on Symbolic and Algebraic Computation, Portland, Oregon, USA*, 1989.
24. Christine Paulin-Mohring. Inductive definitions in the system COQ. In *Typed Lambda Calculi and Applications*, pages 328–345. Springer Verlag, 1993. LNCS 664.
25. Masahiko Sakai and Keiichirou Kusakari. On dependency pairs method for proving termination of higher-order rewrite systems. *IEICE-Transactions on Information and Systems*, E88-D (3):583–593, 2005.
26. Daria Wałukiewicz-Chrzaszcz. Termination of rewriting in the Calculus of Constructions. In *Proceedings of the Workshop on Logical Frameworks and Meta-languages, Santa Barbara, California*, 2000. Satellite workshop of LICS'2000.