Termination of rewriting in the Calculus of Constructions

Daria Walukiewicz-Chrząszcz

LRI, Université de Paris-Sud,
91405 Orsay, France
daria@lri.fr

June 5, 2000

Extended abstract. Full version available at http://www.lri.fr/~daria/papers

1 Introduction

The aim of this paper is to incorporate a rewriting mechanism to the Calculus of Constructions (CC) and present a method that proves strong normalization of the combined calculus.

Practical motivations of our work concern proof assistants based on the Calculus of Constructions and their extension with rewriting-style function definitions. Defining functions by rewriting, instead of `Fix` and `Cases` (if we take Coq [3] as an example), is not only easier but also more powerful, as it can enrich significantly the conversion relation of the system. And the richer the conversion relation is, the more compact and easier to automate the proofs in the system are.

In this respect, our paper contributes to the work on the separation between deduction and computation. Computations are steps that can be mechanically performed and reproduced (like rewriting), and therefore there is no need to keep them in the proof (they are just mentioned as a step of conversion). On the other hand, the reasoning consist of conscious choices of the logical inference rules to be applied.

Incorporating rewriting to the Calculus of Constructions is not straightforward, because the definition of rewriting and the definition of the typing system turn out to be mutually dependent. Another problem is the logical consistency of the resulting calculus. To address this question one has to give a termination criterion, that is satisfied by a set of rewriting rules only if the relation generated by these rules and the beta rule is terminating. Ideally, this criterion should be an automatic decision procedure, running in reasonable time and strong enough to accept the rules which are already known to be terminating, like the elimination rules for inductive types (also called recursor rules).

The General Schema and the HORPO are the two known methods for automatically proving strong normalization of rewriting in the higher-order setting.

The General Schema was originally defined in [9] in the context of simply typed lambda calculus. It was later used in [1, 2] where higher-order rewriting in the Calculus of Constructions was introduced for the first time. The authors restricted themselves to some predefined, basic types and function symbols of algebraic types. The latter result was further extended in [4] by strengthening the General Schema and adding a powerful mechanism, called the Computable Closure, designed to use the information offered by Tait-Girard's reducibility candidates (the method on which are based termination proofs in this context). This mechanism allowed to take care of complex rules as the recursor on Brouwer's ordinals.

Another method for proving strong normalization of higher-order rewriting is Higher Order Recursive Path Ordering (HORPO), which is an extension of the first order RPO [6]. This reduction ordering was presented in [10], in the context of the simply typed lambda calculus. Due to the recursive structure of the HORPO, this method is inherently more powerful than the General Schema, although both methods bear some similarities.

In this paper we extend the latter method to the typing discipline of the Calculus of Constructions. We consider a Calculus of Constructions with constants, representing type constructors, constructors and function symbols, and use HORPO to control rewriting. Type constructors and constructors are used to build inductive types, function symbols serve to define rewriting. For technical reasons every function symbol has two kinds of arguments: parameters and other arguments. The division of arguments is correct if it satisfies star dependency condition. The point of division has also an influence (impact) on the form of the rules accepted by HORPO. Star dependency condition and its interaction with HORPO are the most important concepts of this paper.

Compared to [4], in our system all constants may have dependent types and consequently we can handle the rewriting of dependently typed function symbols. We can define most small strictly positive inductive types (by means of type constructors, constructors and some conditions on them) and write their elimination rules as rewrite rules accepted by HORPO. For the moment we cannot accept elimination rules for inductive types with functional arguments (like Brouwer's ordinals) but we hope to do this very soon by adapting ideas from [4] to our notion of computable closure. It is important to stress that, just like all previously mentioned papers, we consider only object level rewriting, that is rewriting that doesn't create types (strong elimination is out of the scope of this paper).

As in [4], the proof of strong normalization of our calculus is done by using a typed version of the reducibility candidates method due to Coquand and Gallier [5].

Before we go into details, there are two more points to be explained. The first one concerns the status of inductive types. There are no inductive types in the Calculus of Constructions (contrary to the Calculus of Inductive Constructions), which is the starting point of our work. As we have already said, in our calculus we are able to simulate inductive types and their elimination rules by imposing some positiveness conditions on types of constructors and by using rewriting. But inductive types are not necessary to define rewriting in the Calculus of

Constructions. One can perfectly imagine the system without the conditions mentioned above, where constructors are just function symbols that do not rewrite. We decided to present the version with inductive types, that is with positiveness conditions on types of constructors, because these conditions allow us to accept some rules that would not be accepted if constructors were just constants. Another reason is that adapting HORPO and the proof of strong normalization to the special role of constructors was not straightforward.

The second point to be discussed concerns the system for which we prove strong normalization. Instead of proving strong normalization for every set of rules accepted by HORPO, we do it for HORPO itself. More precisely, we mix the Calculus of Constructions with rewrite relation generated by all valid HORPO judgments and show that the resulting calculus is strongly normalizing. Termination for a particular rewrite system accepted by HORPO follows easily.

2 Preliminaries

2.1 Terms

We assume given an infinite set $\mathcal{V}ar$ of variables made of two disjoint infinite subsets $\mathcal{V}ar^{\Box}$ (big variables) and $\mathcal{V}ar^{\star}$ (small variables) and a set of constants divided into type constructors $\mathcal{TC}$, constructors $\mathcal{CS}$ and function symbols $\mathcal{F}$.

Function and constructor symbols will have their arguments classified as *parameters* and other arguments. We write $\mathcal{F}^{r+n}$ ($\mathcal{CS}^{r+n}$) for the set of function (constructor) symbols taking first r parameters and then n other arguments. We simply write $f \in \mathcal{F}^m$ ($c \in \mathcal{CS}^m$) with $m = r + n$, in case the knowledge of r is not needed. Such m is called the *arity* of f (c). Similarly, $s \in \mathcal{TC}^m$ means that s is a type constructor of arity m.

Definition 2.1 *(Pseudoterms)*

$$a ::= x \mid \star \mid \Box \mid \lambda x\!:\!a.a \mid (x\!:\!a).a \mid a\, a \mid s(a_1, \ldots a_n) \mid c(a_1, \ldots a_n) \mid f(a_1, \ldots a_n)$$

The pseudoterm $u\, v$, written also $@(u, v)$, denotes the application of u to v. In the sequel we will also use terms of the form $@(u, v_1, \ldots v_n)$. If $a = (\ldots((u\, v_1)v_2)\ldots v_n)$ then $@(u, v_1, \ldots v_n)$ is a *left-flattening* of a. Note that u may still be an application. The pseudoterm $(x\!:\!u).v$ denotes a product (traditionally it is written $\Pi x : u.v$). If x does not occur in v we write it also $u \to v$.

Pseudoterms are identified with finite labeled trees by considering λx and Πx as binary function symbols different for every x. The *subterm* of a pseudoterm a at position p is denoted by $a|_p$ and the pseudoterm obtained by replacing $a|_p$ by b is written $a[b]_p$. A subterm at position p is called *constructor subterm* if all symbols on the path from the root down to but not including p are constructors.

The notions of *bound variable* and *free variable* are defined as usual. We work *modulo α-conversion* that is identifying the terms that only differ in their bound variables.

Substitutions are written as in $[a_1/x_1, \ldots a_n/x_n]$ or $[\vec{a}/\vec{x}]$, where for every $i = 1, \ldots n$, a_i is assumed different from x_i. The set $\{x_1, \ldots x_n\}$ is called the domain (dom) of the substitution.

The sequence of declarations $x_1 : a_1, \ldots, x_n : a_n$ may be denoted by $\overrightarrow{x : a}$, if n is irrelevant or clear from the context. Vectors of declarations usually serve to denote environments, but we will sometimes use them to form a long product (shortening $(x_1 : b_1) \ldots (x_n : b_n).c$ by $\overrightarrow{(x : b)}.c$). If $\Gamma = x_1 : a_1, \ldots, x_n : a_n$ is an environment then the domain of Γ, denoted by $dom(\Gamma)$, is equal $x_1, \ldots x_n$.

A *declaration of inductive type* is of the form

$$Ind[(p_1 : P_1) \ldots (p_r : P_r)](s : A := c_1 : C_1, \ldots c_m : C_m)$$

where $s \in \mathcal{TC}$ and $c_1, \ldots c_m \in \mathcal{CS}$ (constructors of the inductive type s). The types corresponding to s and $c_1, \ldots c_m$ are obtained by decapsulating the inductive definition: s has a type $(p_1 : P_1) \ldots (p_r : P_r).A$ and every c_i has a type $(p_1 : P_1) \ldots (p_r : P_r).C_i[(sp_1 \ldots p_r)/s]$. Variables $p_1, \ldots p_r$ are called parameters of inductive type s.

As already said in the introduction, we consider only small strictly positive inductive types. By *small* we mean that they are on the type level of the Calculus of Constructions $(A : \Box)$. Positiveness conditions are some syntactic conditions on the form of A and $\vec{C}$. They are given in Appendix A and are the same as in [3, 11].

Example 1 *Inductive definition of polimorhic lists has the form:* $Ind[A : \star]$ $(List : \star := nil : List, cons : A \to List \to List)$ *From this definition* $List : \star \to \star$, $nil : (A : \star)(List\ A)$ *and* $cons : (A : \star)A \to (List\ A) \to (List\ A)$.

A *declaration of function symbol* for an $f \in \mathcal{F}^{r+n}$ is of the form

$$f : (p_1 : P_1) \ldots (p_r : P_r)(x_1 : b_1) \ldots (x_n : b_n).c$$

Additionally, types of constructors and function symbols are subject to star dependency condition, which is described below (see Subsection 2.2).

A *signature* Σ is a (possibly dependent) sequence of declarations of inductive types and function symbols such that we can deduce $\Sigma\ ok$ in the following system:

$$\textbf{(empty)}\quad \frac{}{\emptyset\ ok}$$

$$\textbf{(fun)}\quad \frac{\Sigma\ ok \qquad \Sigma \vdash_{CC} (p_1 : P_1) \ldots (p_r : P_r)(x_1 : b_1) \ldots (x_n : b_n).c : \star}{\Sigma, f : (p_1 : P_1) \ldots (p_r : P_r)(x_1 : b_1) \ldots (x_n : b_n).c\ \ ok}$$

$$\textbf{(ind)}\quad \frac{\Sigma\ ok \qquad \Sigma, \Gamma_p \vdash_{CC} A : \Box \qquad (\Sigma, \Gamma_p, s : A \vdash_{CC} C_i : \star)_{i=1,\ldots m}}{\Sigma, Ind[\Gamma_p](s : A := c_1 : C_1, \ldots c_m : C_m)\ \ ok}$$

where $\vdash_{CC}$ means that we verify the corresponding typing judgment in the usual Calculus of Constructions, treating all constants from $\mathcal{F} \cup \mathcal{TC} \cup \mathcal{CS}$ as variables and using the decapsulated types for type constructors and constructors appearing in Ind declarations.

The definition of the signature makes it clear that we are interested only in small inductive types $(A : \Box)$ and object level rewriting ($f(\vec{a})$ is always an object, $f(\vec{a}) : A : \star$ for some A).

2.2 Star dependency

A type $(p_1 : P_1) \dots (p_r : P_r).(x_1 : b_1) \dots (x_n : b_n).c$, corresponding to a symbol from $\mathcal{F}^{r+n}$ or $\mathcal{CS}^{r+n}$, satisfies the *star dependency condition* if

$$\forall \alpha \in \{b_1, \dots b_n, c\} \;\; \forall y \in \mathcal{V}ar^{\Box} \;\; y \in \mathcal{FV}(\alpha) \Rightarrow y \in \{p_1, \dots p_r\}$$

This condition may be seen as a generalization of the requirement that all variables corresponding to nonparameter arguments ($\vec{x}$ in the example above) belong to $\mathcal{V}ar^{\star}$.

Example 2 *The type of map* $\in \mathcal{F}^{2+2}$,

$$(A : \star)(B : \star)(f : A \to B)(l : List\ A)(List\ B)$$

satisfies star dependency condition.

If we supposed that map $\in \mathcal{F}^{1+3}$ *then its type would not satisfy star dependency condition, as the type of f depends on the big variable B that would not be a parameter.*

Of course, if we supposed that map $\in \mathcal{F}^{4+0}$ *than its type would obviously satisfy star dependency condition. On the other hand, this would have a negative impact on the form of the rules which can be accepted by HORPO (every argument of map would have to be a variable; for details see the definition of HORPO Def. 3.2).*

The star dependency condition is crucial to the proof of strong normalization. Together with the form of the HORPO ordering, it ensures the computability of the big arguments of a function symbol (whose type is of type $\Box$). This is so, because all big arguments appear at parameter positions and parameters are carried over without change in HORPO (see cases 2, 3, 4 of Part I of Def. 3.2).

A similar property must hold in case some function symbol headed term is compared to an application (see case 6 in the definition of HORPO). We call it *application condition* and denote it by *AppCon*. Some intuitions about this condition are given after we define HORPO. The exact definition can be found in Appendix B or in the full version of this paper.

3 HORPO

HORPO (Higher Order Recursive Path Ordering) presented in definition 3.2 below is an adaptation of HORPO [10] to the Calculus of Constructions.

HORPO can compare only terms of equal types. For this reason every judgment of HORPO has a form $\Gamma \vdash u \succ t : A$, where Γ is an environment, u, t terms and A their type. All typing judgments that appear in HORPO are the judgments of the Calculus of Constructions.

To define HORPO one needs a precedence on function symbols and a status for every such symbol. *Status* can be either lexicographic (*Lex*) or multiset (*Mul*), although one can imagine more complicated statuses resulting from mixing *Lex* and *Mul*. Status tells us how to do recursive calls for immediate subterms of the function symbol headed term. *Precedence* is any well-founded quasi-ordering on $\mathcal{F}$ satisfying the properties that equivalent symbols have the same status and that equivalent symbols of status *Lex* have equal arities.

Let us now define three orderings used in the recursive calls : $\succeq^{\mathcal{CS}}$ — the constructor extension of HORPO, $(\succeq^{\mathcal{CS}})_{mul}$ and $(\succeq^{\mathcal{CS}})_{lex}$.

Definition 3.1 *(Constructor extension of HORPO)*
Let Γ be an environment and l, r, A be well-typed terms in Γ, such that $\Gamma \vdash r : A$ holds. We say that r is smaller than l in a constructor extension *of HORPO, denoted by $\Gamma \vdash l \succeq^{\mathcal{CS}} r$, if there exists a constructor subterm l' of l, such that $\Gamma \vdash l' \succeq r : A$ holds.*

We write $\Gamma \vdash l \succ^{\mathcal{CS}} r$ if $\Gamma \vdash l \succeq^{\mathcal{CS}} r$ and not $\Gamma \vdash r \succeq^{\mathcal{CS}} l$.

Orderings $(\succeq^{\mathcal{CS}})_{mul}$ and $(\succeq^{\mathcal{CS}})_{lex}$ are respectively the multiset and lexicographic extensions of $\succeq^{\mathcal{CS}}$.

Compared to RPO [6], HORPO has two new cases allowing to deal with an application and constructor headed term on the right (in Part I) and four cases corresponding to monotonicity rules for application, abstraction, product and constants from $\mathcal{TC} \cup \mathcal{CS}$ (Part II). Moreover, it has more possibilities to deal with subterms of the smaller side, which is described by the following proposition :

$$\mathfrak{P} \;=\; \forall v \in \vec{w} \quad \Gamma \vdash f(\vec{l}) \succ v \;:\; B \quad \text{or} \quad \Gamma \vdash l_j \succeq^{\mathcal{CS}} v \text{ for some } l_j$$

In a recursive call, each subterm of the smaller side, v, is compared either with the whole left hand side $f(\vec{l})$, or with some constructor subterm (by $\succeq^{\mathcal{CS}}$). The choice in $\mathfrak{P}$ prevents the situation when we cannot limit some smaller side subterm because there is no bigger side subterm of the same type (recall that HORPO compares only terms of equal types).

Definition 3.2 *(HORPO)*
Let $\geq_{\mathcal{F}}$ be a precedence and suppose that every $f \in \mathcal{F}$ has a status, which can be either lexicographic (Lex) or multiset (Mul). We define $\succ$ which denotes the strict comparison in HORPO.

$$\Gamma \vdash l \succ r \;:\; B \quad \textit{if}$$

$$\Gamma \vdash_{CC} l \;:\; B_1 \qquad \Gamma \vdash_{CC} r \;:\; B_2 \qquad \Gamma \vdash_{CC} B_1 \overset{*}{\leftrightarrow}_{\beta} B_2 \overset{*}{\leftrightarrow}_{\beta} B$$

Part I $\quad \Gamma = (p_1 : P_1) \ldots (p_r : P_r), \Gamma', \qquad l = f(\vec{p}, \vec{u}) = f(l_1, \ldots, l_m),$

$$f : (x_1 : A_1) \ldots (x_{r+n} : A_{r+n}).A \in \mathcal{F}^{r+n} \quad \textit{and}$$

1. *$\Gamma \vdash l_i \succeq^{\mathcal{CS}} r$ for some i, or*
2. *$r = g(\vec{p'}, \vec{w})$, $g \in \mathcal{F}$, $f >_{\mathcal{F}} g$, $\vec{p'} \subseteq \vec{p}$ and $\mathfrak{P}$, or*
3. *$r = g(\vec{p'}, \vec{w}) = g(r_1, \ldots r_m)$, $f =_{\mathcal{F}} g \in Mul$, $\vec{p'} \subseteq \vec{p}$ and $\Gamma \vdash \vec{l}(\succeq^{\mathcal{CS}})_{mul}\vec{r}$, or*
4. *$r = g(\vec{p'}, \vec{w}) = g(r_1, \ldots r_m)$, $f =_{\mathcal{F}} g \in Lex$, $m = n$, $\vec{p'} \subseteq \vec{p}$, $\Gamma \vdash \vec{l}(\succeq^{\mathcal{CS}})_{lex}\vec{r}$ and $\mathfrak{P}$*

5. $r = c(\vec{w})$, $c \in \mathcal{CS}$ and $\mathfrak{P}$, or

6. $@(w_0, \ldots, w_m)$ is a left-flattening of r, $\mathfrak{P}$ and
   - $\Gamma \vdash l \succ w_0 : A[\vec{l}/\vec{x}]$ and $AppCon((\overrightarrow{p : P}, \overrightarrow{x : b} \vdash A), (\Gamma \vdash @(w_0, \ldots, w_m)))$,
   - $\Gamma \vdash l_j \succeq w_0 : A_j[l_1/x_1, \ldots, l_{j-1}/x_{j-1}]$ for some j and $AppCon((\overrightarrow{p : P}, \overrightarrow{x : b} \vdash A_j), (\Gamma \vdash @(w_0, \ldots, w_m)))$.

Part II

1. $l = \lambda x : a.b$, $r = \lambda x : d.e$ and there exists c such that $\Gamma \vdash_{CC} B \overset{*}{\leftrightarrow} (x : a).c$, $\Gamma \vdash_{CC} a \overset{*}{\leftrightarrow} d$ and $\Gamma, x : a \vdash b \succ e \ : \ c$, or

2. $l = (x : a).b$, $r = (x : d).e$ and $\Gamma \vdash_{CC} a \overset{*}{\leftrightarrow} d$ and $\Gamma, x : a \vdash b \succ e \ : \ B$, or

3. $l = l_1 l_2$, $r = r_1 r_2$ and there exist a, b such that $\Gamma \vdash_{CC} B \overset{*}{\leftrightarrow} b[l_2/x]$, $\Gamma \vdash l_1 \succeq r_1 \ : \ (x : a).b$, $\Gamma \vdash l_2 \succeq r_2 \ : \ a$ and at least one of these inequalities is strict, or

4. $l = e(l_1, \ldots l_n)$, $r = e(r_1, \ldots r_n)$, $e : \overrightarrow{(x : b)}.c \in \mathcal{TC} \cup \mathcal{CS}$, for all i $\Gamma \vdash l_i \succeq r_i \ : \ b_i[\vec{l}/\vec{x}]$ and at least one of these inequalities is strict.

The definition is recursive and its well-foundeeness follows from the fact that recursive calls operate on syntactically smaller terms.

Part I is the heart of HORPO, as its cases 1, 2, 3, 4 correspond to the RPO. In Part I the left hand side is always a function symbol headed term. The right hand side can be either a function symbol headed term (cases 2, 3, 4), a constructor headed term (case 5) or an application (case 6). We can also choose a recursive comparison with a constructor subterm of an immediate subterm of the left hand side (case 1). In case 6 the appropiate left-flattening has to be chosen non-deterministically. This non-determinism prevents the situation when we cannot continue with the recursive comparison because of type incompatibility (HORPO compares only terms of the same type).

What is really new in our HORPO it is the use of parameters of function symbols. In Part I the left hand side has always different variables $\vec{p}$ on its parameter positions, and if the right hand side is also a function symbol headed term then its arguments on parameter positions must be included in $\vec{p}$. This property guarantees that during the rewriting with such a rule, we know exactly all big arguments of the right hand side (as they may occur only at parmeter positions and parameters are passed unchanged) what is very important in the proof of strong normalization.

For the same reason we use application condition in case 6. It says roughly that $f(\vec{p}, \vec{u})$ can be compared with $@(t_0, t_1, \ldots t_m)$, only if all big terms within $t_1, \ldots t_n$ belong to $\vec{p}$. The exact definition of $AppCon$ can be found in Appendix B or in the full version of this paper.

Example 3 *Let $map \in \mathcal{F}^{2+2}$ be of type $(A : \star)(B : \star)(A \to B)(List\ A) \to (List\ B)$ and let $map \in Mul$. The rule*

$$C : \star, D : \star, f : C \to D \vdash map(C, D, f, nil\ C) \to nil\ D : List\ D$$

is accepted by case 5 of HORPO. To accept the rule

$$C:\star, D:\star, f:C\to D, a:C, l: List\ C \vdash map(C,D,f,cons(C,a,l)) \to \\ cons(@(f,a), map(C,D,f,l)) : List\ D$$

we have to first use case 5 and then find some terms to compare recursively with $@(f,a)$ *and* $map(C,D,f,l)$*. For the former, we have* $map(C,D,f,cons(C,a,l)) \succ map(C,D,f,l) : List\ D$ *by case 3 of HORPO and the fact that* $cons(C,a,l)$ *is bigger than* l *in* $\succ^{\mathcal{CS}}$*. Unfortunately, there is no term to compare with* $@(f,a)$ *of type* D*, since the only constructor subterm of the left hand side of type* D *is* a *(which is not enough to compare with* $@(f,a)$*). This problem will be solved in Section 5.1 where the notion of computable closure is introduced.*

Example 4 *Suppose that map is a symbol from the previous example. The rule*

$$C:\star, l: ListC \vdash map(C,C,\lambda x\!:\!C.x, l) \to l$$

cannot be accepted by HORPO because the arguments at parameter positions are not distinct variables.

Our formulation of HORPO is not stable by instantiation. The reason is that when comparing $\Gamma \vdash f(\vec{u}) \succ t\ :\ A$, HORPO requires that arguments of f at parameter positions are different variables. Consequently if we take a substitution that equalizes two parameters then the result wouldn't be accepted by HORPO. But it is not a problem in our framework. Regarding the proofs, we will just use the closure by instantiation and context of HORPO (the rewrite relation generated by HORPO).

4 CC+H

CC+H is the result of mixing together the judgments of the Calculus of Constructions and of HORPO. Except for the last two, all the rules are those of the Calculus of Constructions.

The rule ***(rew)*** defines rewriting as a closure by substitution and context of valid HORPO judgments. It works almost like in the first order rewriting, but we have to take care of typability.

Let a be a term we want to rewrite at position p with the rule $G \vdash l \succ r\ :\ T$ and substitution θ. Because of λ and Π symbols we need to allow θ to substitute the free variables in l by terms in which variables bound above p in a may occur. Let us call the environment consisting of these variables together with their types $\mathcal{PV}(a,p)$ (for path variables). For example, if $a = \Pi y : Nat.f(y, \lambda x\!:\!Nat.g(x))$ then $\mathcal{PV}(a, 222) = y : Nat, x : Nat$ (note that $a|_{222} = g(x)$).

Assuming moreover that a is well-typed in the environment Γ, θ must substitute variables from G (the environment of the HORPO judgment) by terms that are well typed in the environment $\Gamma, \mathcal{PV}(a,p)$. Such a substitution θ is written $\theta : G \to \Gamma, \mathcal{PV}(a,p)$ and is called a well-typed substitution.

The other difference of our calculus with respect to the usual CC is the conversion rule ***(conv)*** which now incorporates not only beta but also the rewrite relation generated by HORPO.

Although the following two definitions are mutually dependent we give them separately for the sake of clarity.

Definition 4.1 *(Well-typed substitution)*
A substitution θ is a well-typed substitution from Δ_1 to Δ_2, *denoted by* $\theta : \Delta_1 \to \Delta_2$, *if for all* $x_i \in dom(\Delta_1)$ *we have* $\Delta_2 \vdash x_i\theta : A_i[x_1\theta/x_1 \ldots x_{i-1}\theta/x_{i-1}]$, *provided* $\Delta_1 = x_1 : A_1 \ldots x_n : A_n$.

Definition 4.2 *(CC+H)*
Given a precedence and status function for HORPO, CC+H is defined as follows:

$$\textbf{\textit{(ax)}}\quad \frac{}{\vdash \star : \Box}$$

$$\textbf{\textit{(var)}}\quad \frac{\Gamma \vdash a : p}{\Gamma, x : a \vdash x : a} \qquad (x \in \mathcal{V}ar^p \setminus dom(\Gamma), p \in \{\star, \Box\})$$

$$\textbf{\textit{(weak)}}\quad \frac{\Gamma \vdash a : b \qquad \Gamma \vdash c : p}{\Gamma, x : c \vdash a : b} \qquad (x \in \mathcal{V}ar^p \setminus dom(\Gamma), p \in \{\star, \Box\})$$

$$\textbf{\textit{(const)}}\quad \frac{\Gamma \vdash a_i : b_i[a_1/x_1, \ldots, a_{i-1}/x_{i-1}]}{\Gamma \vdash e(a_1, \ldots, a_n) : c[a_1/x_1, \ldots, a_n/x_n]} \qquad (e : (x_1 : b_1) \ldots (x_n : b_n).c \in \Sigma)$$

$$\textbf{\textit{(abs)}}\quad \frac{\Gamma, x : a \vdash b : c \qquad \Gamma \vdash (x{:}a).c : p}{\Gamma \vdash \lambda x{:}a.b : (x{:}a).c} \qquad (x \notin dom(\Gamma), p \in \{\star, \Box\})$$

$$\textbf{\textit{(app)}}\quad \frac{\Gamma \vdash a : (x{:}b).c \qquad \Gamma \vdash d : b}{\Gamma \vdash a\, d : c[d/x]}$$

$$\textbf{\textit{(prod)}}\quad \frac{\Gamma \vdash a : p \qquad \Gamma, x : a \vdash b : q}{\Gamma \vdash (x{:}a).b : q} \qquad (x \notin dom(\Gamma), p, q \in \{\star, \Box\})$$

$$\textbf{\textit{(rew)}}\quad \frac{G \vdash l \succ r : T \qquad a|_p = l\theta \qquad \theta : G \to \Gamma, \mathcal{PV}(a,p)}{\Gamma \vdash a[l\theta]_p \succ\!\succ a[r\theta]_p}$$

$$\textbf{\textit{(conv)}}\quad \frac{\begin{array}{cc}\Gamma \vdash a : b & \Gamma \vdash b' : p \\ \Gamma \vdash b(\to_\beta \cup \succ\!\succ)^* b' \quad or & \Gamma \vdash b'(\to_\beta \cup \succ\!\succ)^* b\end{array}}{\Gamma \vdash a : b'} \qquad (p \in \{\star, \Box\})$$

In the rules above, $\Gamma \vdash a \succ\!\succ b$ stands for the closure of HORPO by instantiation and context, $(\to_\beta \cup \succ\!\succ)$ for the sum of beta and rewrite relations and $(\to_\beta \cup \succ\!\succ)^*$ for its reflexive and transitive closure.

Accordingly to the definition of rewriting given in the beginning of this section one would expect $\Gamma \vdash a : A$ among the premises of ***(rew)***. But this cannot be done, because of some technical details in the proof of subject reduction property. Once we prove this property the system given above is equivalent (derives the same typing judgments) to the system where $\Gamma \vdash a : A$ is a premise of ***(rew)*** and the rule ***(conv)*** is defined for one step of beta or rewrite relation.

For simplicity reasons we write $\Gamma \vdash a \to b$ for $\Gamma \vdash a(\to_\beta \cup \succ\!\succ)b$. We also use the notation $\to^\Gamma$ for this relation. Note that the superscript Γ is necessary, as the environment is the internal part of the definition of this relation.

Most metatheoretical properties of CC are still valid in CC+H. This is true, for example, for substitution property, stripping, classification and subject reduction. For more details see the full version paper.

5 Strong normalization of CC+H

To prove the strong normalization of $\rightarrow^{\Gamma}$, we use the well known method of "reducibility candidates". Its untyped version was used in several ways to prove strong normalization of the usual Calculus of Constructions, but rewriting forces us to use sets of well-typed terms for candidates, as introduced in [5].

The proof follows the standard scheme. We first define candidates and interpretation of types and we show that every interpretation is a candidate. Compared to [5] we have to add the case covering rewriting to the definition of candidates and give the definition of interpretation for inductive types.

According to the standard scheme, one then shows the main lemma, saying that every term belongs to the interpretation of its type, and concludes that all terms are strongly normalizing. In our case there are three new kinds of terms for which we have to show the main lemma. Type constructor and constructor headed terms belong to corresponding interpretations by the definition of interpretation of inductive types. The case of function symbol headed terms is much more delicate, as these terms may rewrite and, contrary to beta-reduction, there is not enough information hidden in the interpretation to solve this case easily.

It turns out that the only way to show that the function symbol headed term $f(\vec{t})$ belongs to the interpretation of its type is to verify all its reducts. This is done by inspecting all HORPO cases and reasoning by induction on $>_{\mathcal{F}}$ (the precedence) and $(\rightarrow^{\Gamma})_{stat}$ (reductions from $\vec{t}$, with *stat* the status of f). This kind of lemma (called Fun Lemma in our paper) and induction was already present in [10], but the fact that function symbols have dependent types makes the proof much more involved in our case.

To explain the problem suppose that $f(\vec{t})$ rewrites to $g(\vec{u})$ in HORPO, with $f : (\overrightarrow{x : b}).c$ and $g : (\overrightarrow{y : d}).e$. We have to show that $g(\vec{u})$ belongs to the interpretation of type of $f(\vec{t})$. In fact, the interpretation of the type of $f(\vec{t})$ is the interpretation of c in the presence of the substitution $[\vec{t}/\vec{x}]$ and some ζ which assigns candidates to those $\vec{x}$ that belong to $\mathcal{V}ar^{\Box}$. Let us denote it by $I(c)_{[\vec{t}/\vec{x}],\zeta}$.

Considering $g(\vec{u})$, all one could get from the induction hypothesis are statements of the form $g(\vec{u}) \in I(e)_{[\vec{d}/\vec{y}],\xi}$ for some ξ. Here, the problem arises: how to find the right ξ in order to prove $I(c)_{[\vec{t}/\vec{x}],\zeta} = I(e)_{[\vec{d}/\vec{y}],\xi}$? It is solved thanks to the star dependency condition and the form of the ordering HORPO. The substitution ξ is just ζ, as all big $\vec{y}$ have to be parameters and the parameters of the right hand side are included in the parameters of the left hand side. The equality of interpretations is true by the Small Substitution Lemma, which says that the interpretations of two types are equal if these types are unifiable by a small substitution (i.e. whose domain is included in $\mathcal{V}ar^{\star}$).

For the details and proofs we refer the reader to the Fun Lemma and Small Substitution Sections in the full version of this paper.

5.1 Computable closure

Computable closure was first defined in [4] to strengthen the General Schema and than it was used in [10] as a part of HORPO. The role of computable closure is to construct new terms from the given ones without losing their reducibility property (or rather the property that the term belongs to the interpretation of its type).

Computable closure can be added to our definition of HORPO without changing the proof scheme given above. The reason is that computable closure is a mechanism in some sense orthogonal to HORPO; it strengthens HORPO by using closure properties of candidates without changing the essence of HORPO comparison.

To incorporate computable closure to HORPO definition one has to add it to the proposition $\mathfrak{P}$

$$\mathfrak{P} \;=\; \forall v \in \vec{w} \quad \Gamma \vdash f(\vec{l}) \succ v \,:\, B \;\text{ or }\; \Gamma \vdash l_j \succeq^{\mathcal{CS}} v \text{ for some } l_j \;\text{ or }\; v \in CC_f(\vec{l})$$

and to the first case of the first part of HORPO, which now becomes

$$r \in CC_f(\vec{l}) \text{ or } \Gamma \vdash l_i \succeq^{\mathcal{CS}} r \text{ for some } i$$

Without computable closure, to limit v an immediate subterm of the right hand side, we could use the comparison either with the whole left hand side $f(\vec{l})$, or with one of its immediate subterms (or their constructor subterms). The role of the computable closure is to accept v if it results from simple transformations of $\vec{l}$ (like, for example, applying l_{j1} to l_{j2}). Decomposition of a constructor headed term (getting a_i from $c(\vec{a})$) or application of a function symbol h to its arguments $\vec{t}$ are also examples of transformations computable closure can do (the application of the function symbol is allowed only if $(h, \vec{t})$ are smaller than $(f, \vec{l})$ in $(>_{\mathcal{F}}, (\succeq^{\mathcal{CS}})_{stat})$ which is included in the ordering used to do induction in Fun Lemma).

Example 5 *In the example 3 one couldn't find a term to compare with $@(f, a)$. Thanks to computable closure in the definition of $\mathfrak{P}$ it suffices to show that $@(f, a)$ belongs to $CC_{map}(C, D, f, cons(C, a, l))$. But this follows from the application case of computable closure and the fact that $f \in CC_{map}$ and $a \in CC_{map}$ (by constructor decomposition).*

For more detailed presentation of computable closure and its properties we refer the reader to the Computable Closure Section in the full version paper.

6 Examples

In practice, we do not really consider the system given in definition 4.2. Instead, the user gives some rules expressing the meaning of the function symbol and we want to be sure that the Calculus of Constructions with the conversion enriched by the rewrite relation induced by these rules is terminating. Fortunately, the strong normalization property of such a system is a consequence of the strong normalization of CC+H, if there is a precedence and status for the symbols from $\mathcal{F}$ such that every rule $\Gamma \vdash l \to r : T$ verifies $\Gamma \vdash l \succ r : T$. This follows from the fact that the system based on user-defined rules is a subsystem of CC+H. Before we give some examples of dependently typed rules, let us mention the fact that all positive examples given in [10] are also accepted by our HORPO.

Example 6 *Dependent elimination on lists. Let* $list_{dep} : (A : \star)(P : list\ A \to \star)(P\ (nil\ A)) \to ((a : A)(l : list\ A)(P\ l) \to (P\ (cons\ A\ a\ l)) \to (l : list\ A)(P\ l)$ *be a symbol with two parameters and multiset status.*

$$A : \star, P : list\ A \to \star, f_1 : P(nil\ A), f_2 : (a : A)(l : list\ A)(P\ l) \to (P\ (cons\ A\ a\ l)),$$
$$a : A, l : (List\ A) \vdash$$
$$list_{dep}(A, P, f_1, f_2, nil\ A) \to f_1$$
$$list_{dep}(A, P, f_1, f_2, cons\ A\ a\ l) \to @(f_2, a, l, list_{dep}(A, P, f_1, f_2, l))$$

Let us denote by Γ the environment of the rules. The first rule is easily oriented as f_1 is an immediate subterm of the left hand side (case 1 of HORPO).

To verify the second rule, we first use case 6 of HORPO and we are left to show that $\Gamma \vdash f_2 \succeq f_2$, $\Gamma \vdash (cons\ A\ a\ l) \succeq^{\mathcal{CS}} l$, $\Gamma \vdash (cons\ A\ a\ l) \succeq^{\mathcal{CS}} a$ and $list_{dep}(P, f_1, f_2, l) \in CC_{list_{dep}}(P, f_1, f_2, cons\ A\ a\ l)$.

The first two inequalities are trivial. In the third one we really use the constructor extension of HORPO, since $(cons\ A\ a\ l)$ and a have different types. Note that it is the only way to limit this a, as there are no immediate subterms of $list_{dep}(A, P, f_1, f_2, cons\ A\ a\ l)$ of type A. To get the third assertion, we first decompose the constructor $cons$, to get $a, l \in CC_{list_{dep}}$, and then apply $list_{dep}$ to its arguments (it is possible, as $(P, f_1, f_2, cons\ A\ a\ l)(\succeq^{\mathcal{CS}})_{mul}(P, f_1, f_2, l)$).

Example 7 *fold_left and fold_right on lists with length. Let fold_left, fold_right* $: (A : \star)(B : \star)(A \to B \to B) \to B \to (n : nat)(List\ A\ n) \to B \in \mathcal{F}^{2+4}$ *have lexicographic right status and the following rules:*

$$A : \star, B : \star, f : (A \to B \to B), x : B, a : A, n : nat, l : List\ A\ n \vdash$$

$$fold\_left(A, B, f, x, 0, (nil\ A\ 0)) \to x$$
$$fold\_left(A, B, f, x, sn, cons(A, sn, a, l)) \to fold\_left(A, B, f, @(f, x, a), n, l)$$

$$fold\_right(A, B, f, x, 0, (nil\ A\ 0)) \to x$$
$$fold\_right(A, B, f, x, sn, (cons\ A\ sn\ a\ l)) \to @(f, a, fold\_right(A, B, f, x, n, l))$$

Let us call the environment of these rules Γ. The first rules of *fold_left*, *fold_right* are trivially accepted. To orient the second rule of *fold_left* we use the case lex of HORPO and we are led to show $\Gamma \vdash sn \succ^{\mathcal{CS}} n$, $\Gamma \vdash (cons\ A\ sn\ a\ l) \succeq^{\mathcal{CS}} l$ and $@(f, x, a) \in CC_{fold\_left}(A, B, f, x, sn, cons(A, sn, a, l))$, which are easy. The second rule of *fold_right* may be handled much the same way as the second rule of elimination on lists.

A more complex example can be found in Appendix C. It models the function mem, which given an element a and a list l returns the proof that a is in l or that every element of l is different from a.

The remainder of this section is devoted to examples which do not fit in our current framework.

Example 8 *Constructor that does not satisfy star dependency condition.*

$$\begin{aligned} Ind[](StrangeList : \star \ := \ & nil : \ StrangeList, \\ & cons : \ (X : \star)X \to StrangeList \to StrangeList) \end{aligned}$$

or in Coq style:

$$\begin{aligned} & Inductive\ StrangeList : \star := \\ & \quad nil : StrangeList \\ & \quad cons : (X : \star)X \to StrangeList \to StrangeList \end{aligned}$$

In this example star dependency condition is violated, because X is a big variable that appears in the type of another argument of $cons$, but it is not a parameter. In practice types violating star dependency condition are rare; we have searched Coq's contribution files, but we haven't found any.

Example 9 *Elimination on Brouwer's ordinals. Let* ord_{ind} :

$$\begin{aligned} & P : \star, f_1 : P, f_2 : ord \to P \to P, f_3 : ((nat \to ord) \to (nat \to P) \to P), \\ & F : nat \to ord \vdash \\ & \quad ord_{ind}(P, f_1, f_2, f_3, lim(F)) \to @(f_3, F, \lambda y \!:\! nat.\, ord_{ind}(P, f_1, f_2, f_3, @(F, y))) \end{aligned}$$

In the recursive call of ord_{ind} there is a term $@(F, y)$ where y is a variable bound above ord_{ind}. Our computable closure is not sufficient here. However, this example is accepted by the method from [4] thanks to the computable closure using the notion of positive subterm. We hope that extending our computable closure by these positive subterms will enable to orient this rule correctly.

7 Modularity

The result of this paper applies to the situation when we have one, given from the beginning, set of rules. From the practical point of view, it would be very useful to have a modular method of proving termination. Modularity means that rules are not necessarily given at the same time; they come in packages, the next package possibly using the previous ones for typing reasons. Consider for example rules for $plus$ and $append$ (inductive type $list$ corresponds to integer lists with length)

$$\begin{aligned} plus \ &: \ nat \to nat \to nat \\ & \quad plus\ O\ n \to n \\ append \ &: \ (n, m : nat)\ (list\ n) \to (list\ m) \to (list\ (plus\ n\ m)) \\ & \quad append\ O\ m\ nil\ lm \to lm \end{aligned}$$

Unless we have the rule for $plus$ in the conversion relation of the system, the types of the left and right hand side of the rule for $append$ are different. Consequently HORPO would reject this rule.

Our method would be modular if we could iterate the following steps:

1. verification of the new package of rules R_n in HORPO that uses $CC+H_{n-1}$ for typing,

2. construction of the system $CC+H_n$,

3. proof of strong normalization of $CC+H_n$.

It's obvious how to do two first steps; on the contrary we haven't verified yet whether the proof of strong normalization of $CC+H_n$ can be done.

Note however, that if every coming package is verifiable in the Calculus of Constructions, our method is already modular: we just treat all the packages as one big set of rules to be verified in the initial HORPO.

8 Conclusions

We have defined an extension of the Calculus of Constructions by higher-order rewrite rules of possibly dependent types and presented a criterion (HORPO) which guarantees the strong normalization of the resulting calculus.

The restrictions we impose are twofold:

1. We consider the rewriting only at the object-level, and consequently we use only inductive types of star arity.

2. Types of function symbols and constructors of inductive types have to satisfy star dependency condition (see Section 2.2).

The first point is rather a choice than a restriction. Rewriting at the level of types is a difficult question and raises some important technical difficulties, as shown in [12, 7, 8].

It is not clear how important is the second restriction. At present, it is crucial for the proof of normalization and we believe that it will be very difficult to weaken.

References

[1] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalization and confluence in the λ-algebraic-cube. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 406–415, Paris, France, July 1994. IEEE Comp. Soc. Press.

[2] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalization in the algebraic-λ-cube. *Journal of Functional Programming*, 7(6):613–660, November 1997.

[3] B. Barras, S. Boutin, C. Cornes, J. Courant, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, P. Loiseleur, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual Version 6.2*. INRIA-Rocquencourt-CNRS-Université Paris Sud- ENS Lyon, May 1998.

[4] 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.

[5] T. Coquand and J. H. Gallier. A proof of strong normalization for the Theory of Constructions using a Kripke-like interpretation. In G. Huet and G. Plotkin, editors, *Preliminary Proceedings 1st Intl. Workshop on Logical Frameworks, Antibes, France, 7–11 May 1990*, pages 479–497. 1990.

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

[7] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo, 1998. Rapport de recherche INRIA 3400.

[8] Gilles Dowek and Benjamin Werner. Proof normalization modulo. In B. Rues T. Altenkirch, W. Naraschewski, editor, *Types for Proofs and Programs 98*, volume 1657 of *Lecture Notes in Computer Science*, pages 62–77. Springer-Verlag, 1999.

[9] Jean-Pierre Jouannaud and Mitsuhiro Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349–391, February 1997.

[10] 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. IEEE Comp. Soc. Press.

[11] Christine Paulin-Mohring. Inductive definitions in the system COQ. In *Typed Lambda Calculi and Applications*, pages 328–345. Springer-Verlag, 1993. LNCS 664.

[12] Benjamin Werner. *Méta-théorie du Calcul des Constructions Inductives*. Thèse de doctorat, Univ. Paris VII, 1994.

Appendix A: Positiveness conditions in the definition of inductive types

A pseudoterm A is a $\star$-*arity* if $A = \overrightarrow{(x:b)}.\star$. We denote it by $A \in A_*$.

A *type generated by the type constructor* s is a pseudoterm of the form $\overrightarrow{(x:b)}.sa_1 \ldots a_n$. We write $t \in T_s^m$ if t is a type generated by s and the length of $\overrightarrow{(x:b)}$ is m.

The type generated by s, $t = \overrightarrow{(x:b)}.sa_1 \ldots a_n$, satisfies the *positivity condition* if s doesn't occur in any a_i and it occurs *strictly positively* in each domain of product $\vec{b}$.

We say that s occurs *strictly positively* in a pseudoterm b, if $b = \overrightarrow{(x:d)}.e$ and s doesn't occur in $\vec{d}, e$.

An expression $Ind[\Gamma_p](s : A := c_1 : C_1, \ldots c_m : C_m)$ defines *small strictly positive inductive type* if the following conditions hold, provided $\Gamma_p = p_1 : P_1, \ldots p_r : P_r$:

- $s \in \mathcal{TC}^r, \quad \forall i \exists n_i \; c_i \in \mathcal{CS}^{r+n_i}$,
- $A \in A_*$,
- $\forall i \; C_i \in \mathcal{T}_s^{n_i}$.

Appendix B: Application condition

Definition 8.1 *(Application condition)*
Let $G_p = \overrightarrow{p:P}$, G_1 and G be environments and T_0 be a term such that $(G_p)(G_1).T_0$ satisfies star dependency condition. Moreover let μ be a well-typed substitution from G_p, G_1 to G_p, G which is the identity on G_p ($\mu(p) = p$ for $p \in dom(G_p)$) and $@(t_0, \ldots, t_m)$ be a term, such that $G_p, G \vdash t_0 : T_0\mu$ holds.

We say that a pair of terms T_0 and $@(t_0, \ldots, t_m)$ satisfies the application condition, *denoted by $AppCon((G_p, G_1 \vdash T_0), (G_p, G \vdash @(t_0, \ldots, t_m)))$ if $T_0 = (y_1 : A_1) \ldots (y_m : A_m).B$, and for all y_i*

- *$y_i \in \mathcal{V}ar^\star$, or*
- *$y_i \notin (\mathcal{FV}(A_{i+1}) \cup \ldots \cup \mathcal{FV}(A_m) \cup \mathcal{FV}(B))$, or*
- *$y_i \in \mathcal{V}ar^\Box$, $t_i = p_{k_i}$, $(p_{k_i} : P_{k_i}) \in G_p$ and*
  $G_p, G_1, \overrightarrow{(y^\star : A[\vec{p}^\Box/\vec{y}^\Box])} \vdash A_i[\vec{p}^\Box/\vec{y}^\Box] \overset{*}{\leftrightarrow} P_{k_i}$,

provided that $\vec{y}^\Box = \{y_i \in \mathcal{V}ar^\Box \mid t_i \in dom(G_p)\}$, $\vec{y}^\star = \vec{y} \setminus \vec{y}^\Box$ and $[\vec{p}^\Box/\vec{y}^\Box]$ is a substitution which assigns p_{k_i} to y_i if $y_i \in \vec{y}^\Box$ and $t_i = p_{k_i}$.

The fact that $G_p, G \vdash t_0 : T_0\mu$ holds, imposes that types of t_j and $@(t_0, \ldots, t_m)$ are $A_j\mu$ and $B\mu$ respectively. Recall that A_j, B are terms defined in the environment $G_p, G_1, \overrightarrow{y:A}$. Simulating the star dependency condition blindly would force restrict all $\vec{y}$ to be small or irrelevant to the types $A_1, \ldots A_m, B$ (first two cases of the definition above). But it turns out that we can relax this condition and allow some big variable y_i if t_i is a variable from G_p (the second part of the third case is purely technical).

In other words the application condition ensures that every big variable on which $\vec{A}$ and B depend comes from G_p or is eventually substituted by some variable from G_p.

Appendix C: Example of rules accepted by HORPO

In the example given below we will strongly use the definition of computable closure. Since this definition is too big to put it here, let us just informally enumerate some transformations that computable closure can do and that will be used in our example. These are:

- type constructor and constructor composition (getting $s(\vec{a})$ and $c(\vec{a})$ from $\vec{a}$, where $s \in \mathcal{TC}$, $c \in \mathcal{CS}$),
- formation of the product (getting $(x:A).B$ from A and B),
- abstraction over internal variable of computable closure (getting $\lambda x:A.B$ from B, where $x : A$ is an internal variable).

For the exact definition of computable closure, we refer the reader to the full version paper.

Example 10 *Membership in a list. Given an element a and a list l, the result of mem function will be the proof that a is in l or the proof that every element of l is different from a. To formalize this we need to introduce inductive types representing disjunction, equality, false, membership and verification of a predicate by all elements of the list. Definitions of inductive types are written in Coq style. The type $\sim A$ stands for $A \rightarrow False$.*

$$Inductive\ Or[A:\star, B:\star]:\star := left : A \rightarrow Or\ A\ B\ \ |\ right : B \rightarrow Or\ A\ B$$
$$Inductive\ eq[A:\star, x:A] : A \rightarrow \star := refl\_eq : eq\ A\ x\ x$$
$$Inductive\ False : \star :=$$

$$\begin{array}{l} Inductive\ In[A:\star, a:A] : (list\ A) \rightarrow \star := \\ \quad in\_hd : (l:(list\ A))(In\ a\ (cons\ A\ a\ l)) \\ \quad |\ in\_tl : (l:(list\ A))(b:A)(In\ a\ l) \rightarrow (In\ a\ (cons\ A\ b\ l)) \end{array}$$

$$\begin{array}{l} Inductive\ AllS[A:\star, P:A\rightarrow\star] : (list\ A) \rightarrow \star := \\ \quad allS\_nil : (AllS\ P\ (nil\ A)) \\ \quad |\ allS\_cons : (a:A)(l:(list\ A))(P\ a) \rightarrow (AllS\ A\ P\ l) \rightarrow (AllS\ A\ P\ (cons\ A\ a\ l)) \end{array}$$

Let eq_ind, mem and if have the following types and rules (formally the rules should be given with their environment and types, but we decided to omit them as they can be deduced from the type of function symbols):

$$\begin{array}{rl} eq_{ind}: & (A:\star)(a:A)(P:A\to\star)(b:A)(eq\ A\ a\ b)\to(P\ a)\to(P\ b)\in\mathcal{F}^{3+3},\\ mem: & (A:\star)(a:A)\to((B:\star)(c,d:B)(Or\ (eq\ B\ c\ d)\ (\sim eq\ B\ c\ d)))\to\\ & (l:(list\ A))(Or\ (In\ A\ a\ l)\ (AllS\ A\ (\lambda x\!:\!A.\sim eq\ A\ a\ x)\ l))\in\mathcal{F}^{1+3}\ and\\ if: & (A:\star)(a,b:A)(l:list\ A)\to(Or\ (eq\ A\ a\ b)(\sim eq\ A\ a\ b))\to\\ & (Or\ (In\ A\ a\ l)\ (AllS\ A\ (\lambda x\!:\!A.\sim eq\ A\ a\ x)\ l))\to(Or\ (In\ A\ a\ (cons\ A\ b\ l))\\ & (AllS\ A\ (\lambda x\!:\!A.\sim eq\ A\ a\ x)\ (cons\ A\ b\ l)))\in\mathcal{F}^{1+5} \end{array}$$

$$eq_{ind}\ A\ x\ P\ x\ h\ p\to p$$

$$mem\ A\ a\ eqDec\ (nil\ A)\to right\ (allS\_nil\ A\ (\lambda x\!:\!A.\sim eq\ A\ a\ x)\ (nil\ A))$$

$$mem\ A\ a\ eqDec\ (cons\ A\ b\ l)\to if\ A\ a\ b\ l\ @(eqDec,A,a,b)\ (mem\ A\ a\ eqDec\ l)$$

$$if\ A\ a\ b\ l\ (left\ d)\ i\to left\ (eq_{ind}\ A\ a\ (\lambda x\!:\!A.In\ A\ a\ (cons\ A\ x\ l))\ b\ d\ (in\_hd\ A\ a\ l))$$

$$if\ A\ a\ b\ l\ (right\ d)\ (left\ e)\to left\ (in\_tl\ A\ a\ l\ b\ e)$$

$$if\ A\ a\ b\ l\ (right\ d)\ (right\ e)\to right\ (allS\_cons\ A\ (\lambda x\!:\!A.\sim eq\ A\ a\ x)\ b\ l\ d\ e)$$

For HORPO we choose the precedence in which $mem >_{\mathcal{F}} if >_{\mathcal{F}} eq_{ind}$ and where mem has a multiset status. Let us concentrate on rules 2, 3, 4.

After applying case 5 to the second rule we are left to show that $(allS\_nil\ A\ (\lambda x:A.\sim eq\ A\ a\ x)\ (nil\ A))$ belongs to $CC_{mem}(A,a,eqDec,(nil\ A))$. The only difficulty is to check that $(\lambda x:A.\sim eq\ A\ a\ x)\in CC_{mem}$. By the type constructor composition rule we can get $eq\ A\ a\ x$ with a free variable x. To obtain the negation, we should first introduce False to CC_{mem} again by the type constructor composition rule and then use the product rule. We finish this case by abstracting the variable x and applying constructor composition rule for $allS\_nil$.

Since $mem >_{\mathcal{F}} if$ in the third rule, we are left to show that every immediate subterm of if is comparable with some subterm of mem or belongs to its computable closure. Subterms A and a are just subterms of the left hand side and b, l can be compared with $(cons\ A\ b\ l)$ by means of $\succeq^{\mathcal{CS}}$. We will show that $@(eqDec,A,a,b)$ and $(mem\ A\ a\ eqDec\ l)$ belong to the computable closure of mem. Note that we cannot use directly case 3 (mul) of HORPO to compare $(mem\ A\ a\ eqDec\ l)$ with the left hand side, as the types do not match. By the definition of computable closure, terms A, a, $eqDec$, b and l belong to CC_{mem}. By the application rule we get $@(eqDec,A,a,b)\in CC_{mem}$, by the recursive call rule we get $(mem\ A\ a\ eqDec\ l)\in CC_{mem}$. The application of the recursive call rule is correct as $(A,a,eqDec,(cons\ A\ b\ l))(\succeq^{\mathcal{CS}})_{mul}(A,a,eqDec,l)$.

For the fourth rule we first apply case 5 of HORPO and we are led to show that eq_{ind} applied to its arguments belongs to the computable closure of if. Obviously CC_{if} contains A, a, b, d as they are either left hand side immediate subterms or its constructor subterms. We will show that $(\lambda x:A.In\ A\ a\ (cons\ A\ x\ l))$ and $(in\_hd\ A\ a\ l)$ also belong to CC_{if}. For the latter it is sufficient to use constructor composition rule, for the former we have to first use constructor composition rule to built $(cons\ A\ x\ l)$ with a free variable x, then type constructor composition rule to have $(In\ A\ a\ (cons\ A\ x\ l))$ and finally abstraction rule over x. We complete the proof by applying precedence rule for eq_{ind}.

The acceptance proofs for other rules are similar.